Machine Learning Based Mapping of Data and Streaming Parallelism to Multi-cores

Zheng Wang

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2011

Abstract

Multi-core processors are now ubiquitous and are widely seen as the most viable means of delivering performance with increasing transistor densities. However, this potential can only be realised if the application programs are suitably parallel. Applications can either be written in parallel from scratch or converted from existing sequential programs. Regardless of how applications are parallelised, the code must be efficiently mapped onto the underlying platform to fully exploit the hardware's potential.

This thesis addresses the problem of finding the best mappings of data and streaming parallelism—two types of parallelism that exist in broad and important domains such as scientific, signal processing and media applications. Despite significant progress having been made over the past few decades, state-of-the-art mapping approaches still largely rely upon hand-crafted, architecture-specific heuristics. Developing a heuristic by hand, however, often requires months of development time. As multi-core designs become increasingly diverse and complex, manually tuning a heuristic for a wide range of architectures is no longer feasible. What are needed are innovative techniques that can automatically scale with advances in multi-core technologies.

In this thesis two distinct areas of computer science, namely parallel compiler design and machine learning, are brought together to develop new compiler-based mapping techniques. Using machine learning, it is possible to automatically build high-quality mapping schemes, which adapt to evolving architectures, with little human involvement.

First, two techniques are proposed to find the best mapping of data parallelism. The first technique predicts whether parallel execution of a data parallel candidate is profitable on the underlying architecture. On a typical multi-core platform, it achieves almost the same (and sometimes a better) level of performance when compared to the manually parallelised code developed by independent experts. For a profitable candidate, the second technique predicts how many threads should be used to execute the candidate across different program inputs. The second technique achieves, on average, over 96% of the maximum available performance on two different multi-core platforms.

Next, a new approach is developed for partitioning stream applications. This approach predicts the ideal partitioning structure for a given stream application. Based on the prediction, a compiler can rapidly search the program space (without executing any code) to generate a good partition. It achieves, on average, a 1.90x speedup over the already tuned partitioning scheme of a state-of-the-art streaming compiler.

Acknowledgements

First of all, I want to express my deepest gratitude to my academic advisor, Professor Michael O'Boyle, for his time, ideas, advice and support. His creativity, enthusiasm and optimism have been a constant source of inspiration.

Thanks a lot to all members of the CARD group here at the University of Edinburgh. Thank you for making the group a pleasant place to be.

Thanks a lot to Björn and Georgios with whom I had successful collaborations, resulting in parts of this thesis.

Thanks a lot to everyone who took the time to read over endless drafts of this thesis. In no particular order they are Murray, Paul, Yansong, Nikolas, Hugh, Christophe, Chris, Dominik, Igor, Stephen and Andrew.

Thanks to all my friends in Edinburgh and China, too numerous to mention here.

Special thanks to my Mum, Dad and sister Yi for the unconditional love and support.

Lots and lots of thanks to my wife Ting, for her understanding, tolerance and encouragement. No amount of words could fully convey my gratitude to her.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- Zheng Wang and Michael O'Boyle. "Mapping Parallelism to Multi-cores: A Machine Learning Based Approach". In: *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, February 2009, 75-84.

- Georgios Tournavitis, Zheng Wang, Björn Franke and Michael O'Boyle. "Towards a Holistic Approach to Auto-Parallelization—Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping". In: *ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI)*, June 2009, 177-187.

- Zheng Wang and Michael O'Boyle. "Partitioning Streaming Parallelism for Multi-cores: A Machine Learning Based Approach". In: *19th Intl. Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2010, 307-318.

(*Zheng Wang*)

Contents

Chapter 1

Introduction

For decades, it was possible to improve performance of single-core processors by increasing clock speed and innovative micro-architecture designs. Unfortunately, because of the power and thermal design constraints, these techniques are no longer able to improve the performance of new processors. In order to make use of the excess transistors (as a result of Moore's Law) without violating design constraints, computer architects have designed multi-core processors by placing multiple processing units (i.e. cores) on a single chip. Because multi-core designs already make up most of the new processor shipments and the number of cores per chip is expected to successively increase in the foreseeable future [Intel, b], multi-core processing is now considered mainstream.

Multi-cores offer the promise of high performance by integrating multiple cores on a single die. To make use of such potential, however, new and existing applications must be written or transformed so that they can be executed in parallel. For this reason, there is an acute need for productive techniques that can help programmers develop parallel applications. As depicted in figure 1.1, the development of parallel applications can be typically broken into three stages. The first stage, *parallelism expression*, focuses on exposing the parallel execution opportunities of the program. This is achieved by identifying a number of parallel tasks of the program (e.g. a number of parallelisable loops or functions) and then presenting the tasks in a suitable form (e.g. a program source). The second stage, *parallelism mapping*, focuses on obtaining an efficient parallel execution on a *particular* machine. It decides how to allocate tasks to threads and how the tasks should be scheduled. In the final stage, *code generation*, the mapping decisions will be converted to a platform-specific binary. This thesis focuses solely on the second stage—parallelism mapping.

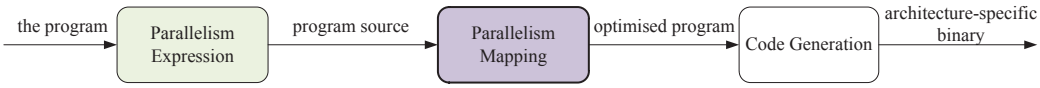


Figure 1.1: The process of developing a parallel application. This process can be typically broken into three stages: parallelism expression, parallelism mapping and code generation. In the first stage, parallel execution opportunities are exposed and then expressed in a suitable form. In the next stage, the expressed parallelism is mapped onto the underlying architecture to efficiently utilise the hardware resource. In the final stage, the mapping decisions are translated into a platform-dependent binary. This thesis focuses solely on parallelism mapping.

Parallelism can be either (semi-) automatically discovered from sequential programs, or manually expressed by programmers using high level programming languages or models. Regardless of how the application is parallelised, once the programmer has expressed this program parallelism in a certain form, the code must be mapped efficiently to the underlying hardware if the potential performance of the hardware is to be realised. Unlike parallelism expression that is largely program dependent, finding the best mapping is highly platform or hardware dependent. The right mapping choice depends on not only the program itself but also the relative costs of communication and computation as well as other hardware costs. Therefore, finding the right mapping is non-trivial.

It is generally agreed that manual parallelism mappings performed by expert programmers result in the most efficient implementations on a given platform, but at the same time this is the most time-consuming and error-prone approach. With a manual approach, programmers are asked to concern themselves with issues of granularity, load-balancing, synchronisation and communication, which are highly dependent on the type of the machine used. Though the manually generated code can be efficient on a particular platform, the parallel implementation is tightly coupled to a specific architecture. To port the application onto a new platform, developers often have to rewrite the code, which comes at a cost of significant investment in terms of time and human effort.

Alternatively, one can develop compiler-based mapping approaches to automatically generate efficient code while ensuring formal correctness of the resulting parallel implementation. Nonetheless, developing a high-quality, compiler-based mapping scheme by hand is difficult too. The difficulty mainly comes from two reasons. First of all, in order to generate an efficient mapping, a conventional compiler often uses an

abstract model to capture the program's behaviour on the target architecture. Whereas computer systems were once easy to model, abstract models can no longer adequately describe today's sophisticated multi-core designs. Second, even if a human could understand enough about the underlying architecture and could afford months of time on developing an accurate enough model, making sure the model works well for a diverse set of programs is a daunting task. As a consequence, a hand-tuned compiler that performs well on one class of programs often fails to produce reasonably good performance on other types of applications [Stephenson, 2006].

Not only is the interaction of parallel applications and architectures difficult to model, their relationship changes over time. Consider the effort that engineers have to spend on (re-) tuning the compiler for subsequent releases of the multi-core processors. For instance, the Intel Xeon processor family, a typical multi-core processor family, has more than 19 types of processors where some processors provide considerably distinct hardware features. It is clear that the task of manually customising a compiler even only for a single processor family is difficult, if not impossible. Because it is extremely hard for compiler designers to keep pace with architecture evolution, the compiler will eventually become out of date and as a result, the hardware potential of the new processor will not be realised. Therefore, current approaches of compiler designs are no longer feasible. What is needed, indeed, are innovative techniques that can automatically scale with the advances of processor technology without being a burden to developers.

Machine learning is one such technique. By automatically *learning from data*, it has the potential to change the way we construct compilers. With machine learning, a compiler automatically learns how to optimise applications instead of relying on hardwired, human-derived heuristics that will inevitably become out of date. As a result, the compiler can adaptively cope with the evolving architecture and with future application domains as well.

1.1 Machine Learning for Compilation

Recently, the use of machine learning to develop optimising compilers has received considerable attention and encouraging results have been reported for sequential programs. Some early experiments have shown that machine-learning-based compilers can perform as efficiently as iterative compilation techniques, which use executions

of many different versions of the input program to perform compilation, but having significantly lower compilation overhead.

Though machine learning is useful in optimising sequential programs, there has been little application on optimising parallel applications. One possible reason may be the inherent complexity of parallel applications and architectures makes it difficult to cast the optimisation target to a machine learning problem. Hence much work remains to be done in this area.

1.2 The Problem

This thesis addresses the issue of parallelism mapping, which is concerned with how to map a parallel program onto a given multi-core architecture so that the mapped program can execute efficiently. In the settings of this thesis, the input of a mapping model is an already parallelised program which is presented as a program source or compiler intermediate code representations. The optimisation goal considered in this thesis is to minimise the execution time of the input program.

Parallelism mapping can be performed manually by the programmer or automatically by a compiler. Given that the number and type of cores of a processor is likely to change from one generation to the next, finding the right mapping for an application may have to be repeated many times throughout an application's lifetime making an automatic technique attractive. This thesis aims to develop automatic and portable compiler techniques.

The main focus of this work is on mapping data and streaming parallelism onto multi-core processors:

- In the case of data parallelism, this thesis addresses three important problems: (a) determining whether a data parallel candidate is profitable to be parallelised; (b) predicting how many threads should be allocated to a profitable candidate; and (c) deciding how a profitable candidate should be scheduled.

- In the case of streaming parallelism, this thesis is concerned with how to partition the input program graph so that an efficient execution can be achieved. The partitioning problem consists of breaking up the input program graph into a number of regions and then allocating the regions into threads.

1.3 Contributions

This thesis presents several novel approaches, based on machine learning, to map parallelism onto multi-core processors. By "learning" from empirical observations, a mapping model can be automatically built for any given platform. This thesis demonstrates that using machine learning, it is able to improve human-generated heuristics by large margins. By offloading much of the compiler design to an automated process, the work of this thesis makes it possible to generate efficient mapping schemes while drastically reducing the human involvement.

On current complex systems, the only way to know how well a scheme performs is to test it empirically; as such, each proposed scheme is evaluated by using empirical performance observations on real systems with well-established benchmarks.

The following list highlights the major contributions of this thesis.

- Firstly, this thesis presents a method for selecting *profitable* data parallel candidates of a parallel application. The proposed machine-learning-based scheme predicts whether a data parallel candidate is profitable to be parallelised and how a profitable candidate should be scheduled. By learning *off-line* from training programs, the proposed model achieves great performance improvement over classically fixed, platform-specific methods when mapping a *new, unseen* program. Since an input program may contain abundant data parallelism and often not all the specified data parallel candidates are beneficial for parallel execution, identifying the profitability is a first step towards an efficient data parallelism mapping. This automatic approach can be used in a compiler to eliminate unprofitable candidates, so as to avoid the performance penalty associated with the parallel execution of these unprofitable candidates. Chapter 4 presents this work.

- Next, a predictive technique is proposed to determine the parallel configuration, i.e. the number of threads used for execution and the corresponding scheduling policy, for a profitable data parallel candidate. This technique makes predictions for new programs across different input data sets. Experimental results show that the proposed scheme not only adapts to different multi-cores, but also outperforms state-of-the-art heuristics while reducing the compilation overhead by an order of magnitude. Chapter 5 discusses this work.

- Finally, this thesis describes a novel approach to partitioning stream programs based on machine learning. The proposed approach has two major contributions:

1. First, a benchmark generator is developed to overcome the problem of insufficient stream benchmarks, which affects the accuracy of a machine learning model. The benchmark generator allows compiler developers to generate many new, small programs and then to select representative programs as training programs to build an accurate model.
2. Second, a new stream-application-partitioning method is presented. This method achieves great performance improvement over traditionally hard-wired, expert-tuned heuristics. Given with this method, a compiler can apply program-aware mappings for stream programs.

The stream graph partitioning problem is essentially *unbounded*, i.e. the number of partitioning operations is unbounded. No previous work about machine learning in compilers has considered unbounded compilation problems; this thesis is the first to do so. Chapter 6 discusses this work in detail.

1.4 Thesis Outline

The remainder of this thesis is organised as follows.

Chapter 2 introduces multi-core processors, programming languages and parallelism paradigms that are used in this thesis. It also describes related machine learning techniques and the evaluation methodology, which are used throughout this thesis.

Chapter 3 discusses related work. It first reviews prior work on heuristics-based mapping schemes and iterative compilation techniques. Then, it outlines existing work on machine-learning-based compilation. Finally, it describes runtime adaptation approaches that dynamically map and schedule parallelism.

Chapter 4 presents an approach to select profitable parallel candidates of a parallel program. This approach is used to replace the originally fixed, target-specific heuristic of a profile-driven auto-paralleliser [Tournavitis and Franke, 2009]. This chapter shows how the proposed scheme outperforms the hand-crafted profitability evaluation approaches. The work of this chapter is based partially on the work published in [Tournavitis et al., 2009].

Chapter 5 demonstrates that the thread allocation scheme has a significant impact on performance and should not be neglected. It presents two machine-learning-based models to determine the number of threads and the scheduling policy for a data parallel candidate. Using low-cost profiling information, both models make predictions for a given program across multiple input data sets, leading to better performance over state-of-the-art approaches. This chapter is based on the work published in [Wang and O'Boyle, 2009].

Chapter 6 develops a stream graph partitioner to generate efficient partitions of stream applications. It describes how to formulate the optimisation problem so that a machine-learning-based model can be built. This chapter also presents a benchmark generator to generate training examples that can be used for training the graph partitioner. This chapter is based on the work published in [Wang and O'Boyle, 2010].

Chapter 7 summarises the main findings of this thesis, presents a critical analysis of this work and discusses future work.

Appendix A describes the statistical method used in chapter 6 to select representative benchmarks from the generated programs.

Chapter 2

Background

This chapter presents the technical background of this thesis. The first section introduces two types of multi-core designs. Then, section 2.2 describes parallel paradigms and programming languages that are employed in the work of this thesis. The specific machine learning techniques used throughout the thesis are presented in section 2.3. Finally, section 2.4 explains the evaluated methodology used in this thesis and section 2.5 summarises this chapter.

2.1 Multi-cores

This thesis targets modern multi-core processors which come in various flavours. In *homogeneous* multi-core designs all cores are identical; in *heterogeneous* multi-core designs cores can vary in terms of processing power. Multi-cores can also differ in the view of memory offered to threads that run on separated cores. In a shared memory multi-core design, a global, shared memory space is provided to all threads. This means all threads within one application can have a single view of data; that is to say, they can directly operate on data located on other processors. In a distributed memory design, there is no such a shared memory space and instead, each core has its own private memory space. This means a thread can only perform operations on data that are located on local memory and if remote data are required, the thread must explicitly move the data from other processors' memory. This thesis makes no assumption of the type of multi-cores; thereby, the techniques proposed in the thesis can be applied to different multi-core designs.

Experiments were evaluated on two representative multi-core processors: Intel XEON [Intel, b] and IBM CELL Broadband Engine [IBM] processors. Figure 2.1 is

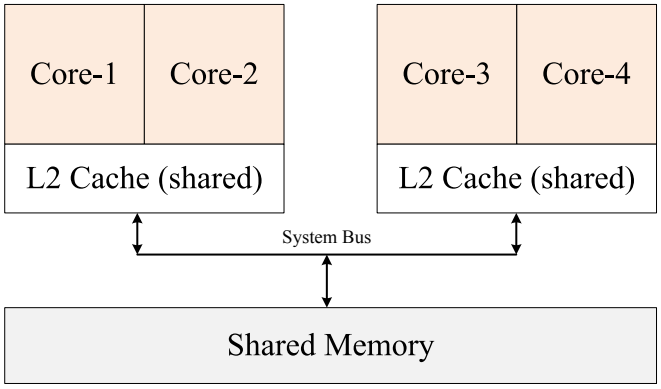


Figure 2.1: Intel XEON, a homogeneous multi-core. All cores within the chip have the same hardware configuration and they share one global memory space.

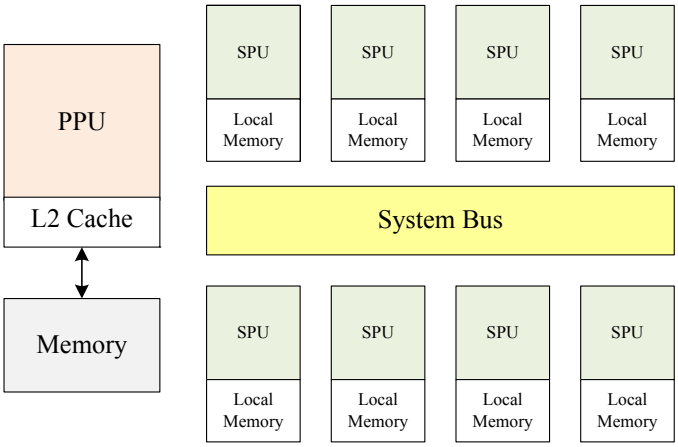


Figure 2.2: IBM CELL, a heterogeneous multi-core. This processor has two different types of cores: one general purpose core, i.e. PPU, and eight accelerators, i.e. the SPUs.

an example of a 4-core Intel XEON processor—a homogeneous and shared memory multi-core. All cores in the Intel XEON design have the same processing capacity and share one global memory space. The CELL processor shown in figure 2.2, on the other hand, is composed of two different types of cores: a Power Processing Unit (PPU) and eight Synergistic Processing Units (SPUs). The PPU is a general purpose processor, while the SPU is a specialised processor optimised for computing intensive tasks. Unlike the PPU, th SPU can only operate on data that are located in its private memory. Hence the CELL processor is a heterogeneous, distributed memory multi-core and data has to be moved in and out the SPU's local memory under software control.

Though the variety of multi-core designs offers flexible options for different application domains, developing parallel applications to fully exploit the potential provided

by the diverse multi-core designs is certainly non-trivial [Patterson and Hennessy, 2008]. Fortunately, there are some common patterns that can be used to parallelise a program. The next section introduces four representative parallelism patterns: data, pipeline, task and streaming parallelism. It also describes two parallel programming languages which can help programmers to apply these patterns in parallel programming practice.

2.2 Parallel Paradigms and Programming Languages

This section first introduces several types of parallel patterns. It then describes two programming languages: OPENMP [Dagum and Menon, 1998] and STREAMIT [Gordon et al., 2002], which are suitable for expressing these patterns and are used in the work of this thesis.

2.2.1 Parallel Paradigms

The first step of parallel programming is identifying parallelism. This focuses on breaking up the problem into a number of tasks[1] so that each task can execute on the parallel machine simultaneously with others. The tasks are intended to execute concurrently but cannot, in many cases, execute independently. This is because there may be data dependences associated with tasks. The term *data dependence* here means that the computation to be performed by one task requires data that are produced by other tasks. Because of data dependences, dependent data must be transferred from one process or thread (that executes a task) to another via communication. Depending on the problem and how the parallel processes are generated and communicate, an application can be parallelised with a number of patterns which are known as parallel paradigms. This section introduces four typical parallel paradigms: data, pipeline, task and streaming parallelism, which are the most related to the work of this thesis.

Data Parallelism

One of the common parallel patterns is data parallelism that can be applied on different granularities such as an instruction level or a loop level. Since this thesis considers exploiting data parallelism at the loop level, “data parallelism” and “data parallel loops”

[1] In this thesis, the term *task* is referred as a set of program instructions (e.g. a loop or a function) that perform some operations.

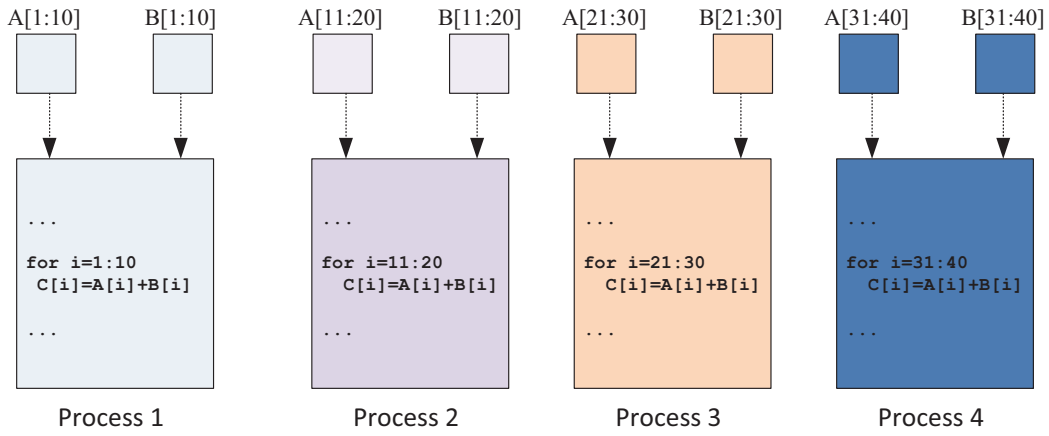


Figure 2.3: Executing a loop in a data parallel fashion. The computing data, i.e. arrays A and B, are distributed across different processes which execute the same code.

```
1    ...
2    while (!EOF(input_file))
3    {
4        r_data = read(input_file);
5        m_data = manipulate(r_data);
6        write(m_data, output_file);
7    }
8    ...
```

Figure 2.4: A program with three operations that can be parallelised with pipeline parallelism.

are used interchangeably throughout the thesis. A data parallel loop means there is no data dependence between one loop iteration and the next. Parallel execution of a data parallel loop can be achieved by distributing different pieces of the computation data across parallel processes (or threads) which execute the *same* code. Under data parallel execution, all processes perform collectively on the same data set (e.g. an array), whereas each process operates on a different partition of the set. The data partitioning is usually achieved by distributing the loop iterations across processes.

An example of exploiting data parallelism is given in figure 2.3. In this case, the computation data of a loop, i.e. arrays A and B, are broken into four sub-sets which are distributed across four parallel processes.

Pipeline Parallelism

When parallelising a program, there may be data dependences that must be respected. Consider the code example given in figure 2.4.The code performs a chain of three

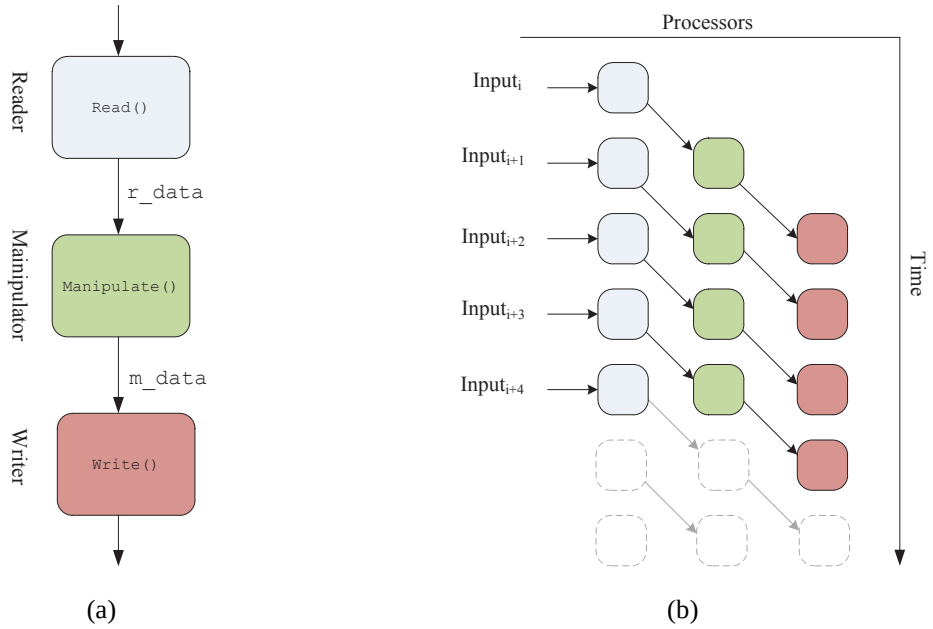


Figure 2.5: The program shown in figure 2.4 is broken into three processes (a) which can execute in parallel (b). Each node represents a process and the arc between two processes represents the communication between them.

operations: (1) reading a chunk of data from a file, (2) manipulating the data and (3) writing the processed data into another file. There are data dependences between two successive operations, e.g. the input of the function *manipulate* is the output of the function *read*. As shown in figure 2.5(a), one can parallelise the program by offloading the three operations into three parallel processes of which each process communicates with its downward process through a data channel (i.e. pipe). Because computation data only flow in one direction between parallel processes, this parallel pattern is called pipeline parallelism.

Figure 2.5(b) shows the pipeline execution of the program. As can be seen from this diagram, when the *reader* process is reading the next data chunk, the *manipulator* process can work on the last data chunk already received and the same goes for the *writer* process; as a result, all processes can execute in parallel.

In a general form of pipeline parallelism, a number of processes perform partial processing of data and then forward the partially processed result to another process down the pipeline for further processing. By forwarding data through the pipeline, processes running with different code can execute in parallel. This is different from data parallelism where all parallel processes execute the same code and no communication is needed between processes.

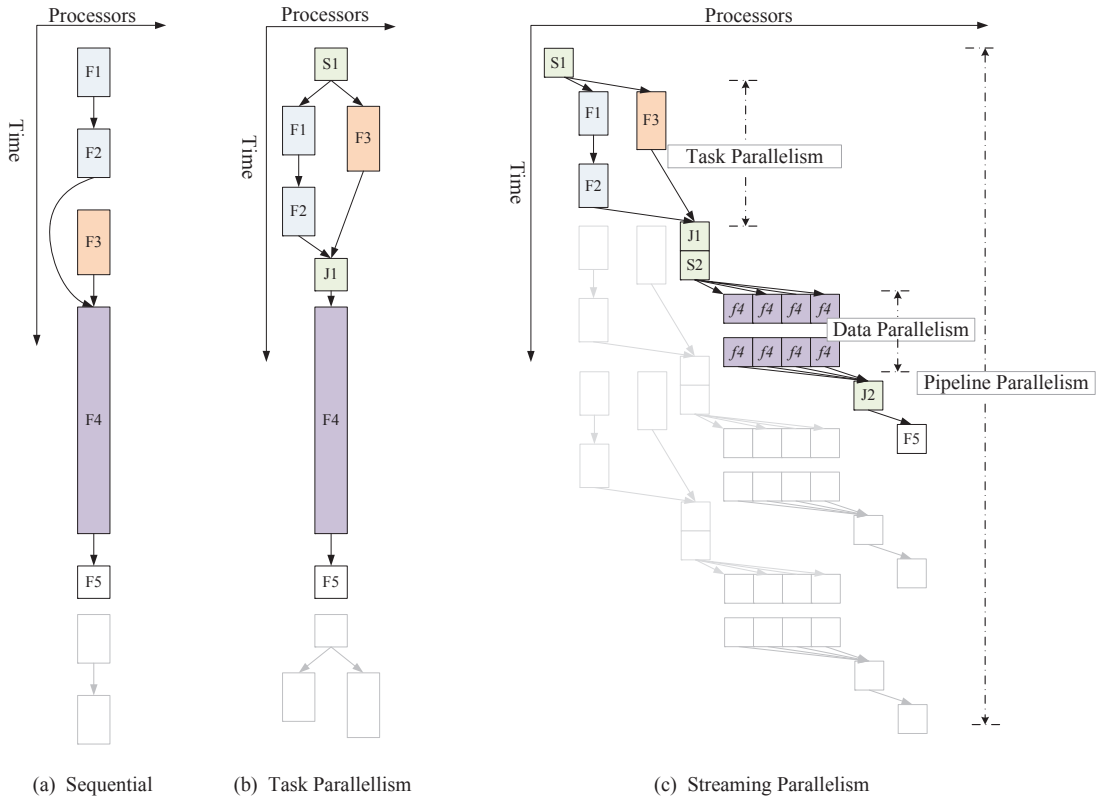


Figure 2.6: Examples of parallel execution of the sequential program (a) with task parallelism (b) and stream parallelism (c). The sequential program repeatedly performs five operations (i.e. functions). Task parallelism is achieved by executing the two independent functions in parallel. Streaming parallelism is achieved by exploiting task, data and pipeline parallelism of the original program.

Task Parallelism

Task parallelism can be viewed as a superset of data and pipeline parallelism, though it is mainly applied at a coarse grain level, e.g. a function level. From a single application's point of view, task parallelism means distributing different code segments (i.e. tasks) across parallel processes. In the general case, the parallel processes can execute independently (which is different from pipeline parallelism) or they have to communicate with others due to data dependences (which is different from data parallelism). Task parallelism is also different from data parallelism in that independent processes may execute different code.

Consider now the sequential execution of a program shown in figure 2.6(a). According to the diagram, *F1* and *F2* can be executed independently with *F3* because the outputs of *F1* and *F2* will never be used by *F3*. Therefore, a task parallel execution

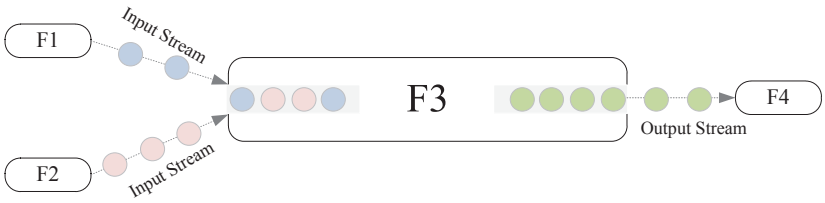


Figure 2.7: An example of how a stream process operates under streaming parallelism. In this example, *F3* reads a number of data items from *F1* and *F2*, performs some computation on the data and passes the processed data to *F4*.

of the program will be similar to figure 2.6(b). In this figure, *S1* spawns three parallel tasks to execute the three functions, where *F1* and *F2* execute in a pipeline parallel fashion and they run independently with *F3*. Parallel executions of the three tasks are synchronised by *J1*.

Streaming Parallelism

Streaming parallelism can be regarded as a generic form of pipeline parallelism. Perhaps the most important characteristic of stream parallelism is that the parallel program operates repeatedly on a stream of data items, e.g. an audio stream. As illustrated by figure 2.7, in streaming parallelism, a parallel task typically performs in a way that—on each execution step—it reads one or more data items from one or more input streams, processes the input data and passes the processed data to one or more output streams. Streaming parallelism is different from pipeline parallelism mainly in two aspects. First of all, in classical pipeline parallelism, each parallel process has only one input and one output while a task in the context of stream parallelism may have multiple inputs and outputs. Second, each process (except for the first process of the pipeline) in pipeline parallelism is dependent on other processes, while some parallel processes in streaming parallelism may be independent processes; so that we can make use of task and data parallelism to execute them in parallel. In other words, streaming parallelism contains task, data and pipeline parallelism.

Figure 2.6(c) is a streaming parallel execution of the sequential program shown in 2.6(a). The majority of tasks have data dependence so that they execute in a pipeline parallel fashion. Some of the tasks, however, are parallelisable with data or task parallelism. For this specific example, data parallelism is achieved by breaking up *F4* into data parallel tasks and task parallelism is achieved by running the different branches of the sequential program (*F1* and *F2*, and *F3*) in a task parallel fashion.

2.2.2 Parallel Programming Languages

To ease the burden of parallel programming, high level parallel programming languages isolate the platform-dependent implementations of parallelism from the specification of parallelism. This is achieved by providing abstractions of the problem domain and hiding details of the underlying platforms. Examples of parallel programming languages include OCCAM [Corp, 1984], High Performance FORTRAN [FORTRAN], OPENMP [Dagum and Menon, 1998], STREAMIT [Gordon et al., 2002], CHAPEL [Chamberlain et al., 2007] and X10 [Saraswat et al., 2007]. In several application domains, e.g. high performance computing and streaming processing, parallel programming languages make the process of developing parallel applications simpler and the code more understandable with respect to serial languages (e.g. C) and libraries (e.g. PTHREAD) [Gordon, 2010]. This section introduces two representative programming languages that are used in this thesis.

OPENMP

```
1  #pragma omp for schedule(static)
2    for (i = iend; i >= ist; i--) {
3      for (j = jend; j >= jst; j--) {
4        for (m = 0; m < 5; m++) {
5          tv[i][j][m] =
6            omega * (  udz[i][j][m][0] * v[i][j][k+1][0]
7                     + udz[i][j][m][1] * v[i][j][k+1][1]
8                     + udz[i][j][m][2] * v[i][j][k+1][2]
9                     + udz[i][j][m][3] * v[i][j][k+1][3]
10                    + udz[i][j][m][4] * v[i][j][k+1][4] );
11       }
12     }
13   }
```

Figure 2.8: Using OPENMP annotations to express data parallelism. This loop is annotated as a data parallel loop with OPENMP clauses. The scheduling policy of the loop is set to static scheduling here.

OPENMP is a parallel programming standard that is widely used for exploiting data parallelism. It provides a set of constructs to express both parallelism and scheduling policies. For example, with OPENMP the programmer can declare a loop as parallelisable by putting the "*#pragma omp for*" clause before the loop header (as illustrated by figure 2.8). The iterations of the annotated loop will be distributed among several *worker* threads when the program executes.

For a data parallel loop that is computed by a number of worker threads, there are four types of scheduling policies available in OPENMP:

1. STATIC: The loop iterations are evenly divided into chunks according to the number of work threads. Each worker thread is assigned a separate chunk.

2. CYCLIC: Each of the loop iterations is assigned to a worker thread in "round-robin" fashion.

3. DYNAMIC: The loop iterations are divided into a number of chunks with a small size. Chunks are dynamically assigned to worker threads on a *first-come, first-served* basis as threads become available.

4. GUIDED: The loop iterations are divided into chunks such that the size of each successive chunk is exponentially decreasing until the default minimum chunk size, 1, is reached.

Figure 2.8 shows a data parallel loop with OPENMP clauses where the loop is declared as a data parallel loop and the scheduling policy is specified as static.

STREAMIT

STREAMIT [Gordon et al., 2002] is a high level programming language based on the Synchronous Data Flow (SDF) model [Lee and Messerschmitt, 1987]. In STREAMIT, computation is performed by *filters* which are the basic computational units. Filters communicate through dataflow channels which are implemented as FIFO queues. STREAMIT provides a simple means of constructing rich hierarchically parallel structures such as *pipeline* and *splitjoin*. A splitjoin structure represents independent parallel execution paths that diverge from a common splitter and merge into a common joiner.

A STREAMIT program may contain three types of coarse-grain parallelism: task, data and pipeline parallelism. Task parallelism refers to different branches in a splitjoin section, where each branch contains a number of filters, and the output of each branch is never used by the other. Data parallelism is achieved by breaking up a "stateless" filter into multiple filters. A filter is "stateless" if there are no data dependences between one execution of the filter and the next. It is the compiler's decision to duplicate "stateless" filters to introduce data parallelism. Pipeline parallelism is understood as multiple chains of filters that are directly connected in a STREAMIT program.

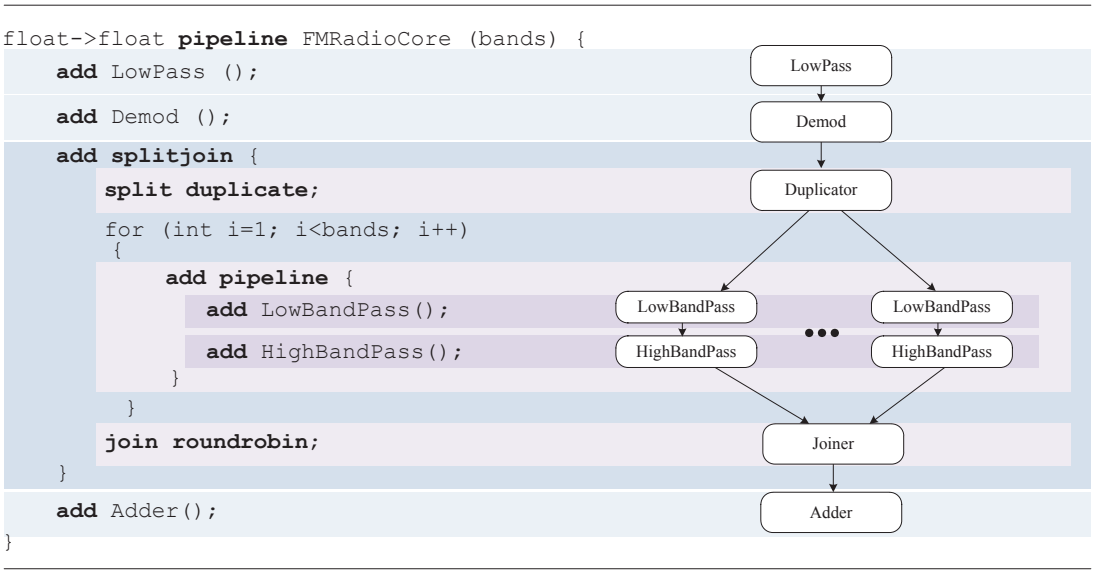


Figure 2.9: A STREAMIT code and its corresponding stream graph. There is a natural mapping between the STREAMIT clauses and the structure of the graph.

Figure 2.9 shows the source code of the STREAMIT *software FM radio* benchmark and its corresponding stream graph. Each node in the stream graph is a task that can execute in data or pipeline parallel fashion and communication between tasks is defined by and restricted to the arcs between nodes. Concurrent task parallelism can be achieved by running the branches of the splitjoin section in parallel.

STREAMIT is a sophisticated language for describing streaming parallelism, which has an accompanying compiler and benchmarks, making it suitable for the work presented in chapter 6.

2.3 Machine Learning Techniques

2.3.1 Terminology

The term *machine learning* refers to the design and implementation of algorithms that allow computer machines to “learn”.

There are two major subdivisions of machine learning techniques. In one, called *supervised learning*, the technique involves learning a function from a set of training data. Each training example in the training data is a pair of input variables and a desired output, $(x_i; y_i)$, where x is an n-dimensional input vector of numerical values and y is the output. The components of the input vector x are called *features* and the input vector x is referred as the *feature vector*. The set of the feature vectors in the training

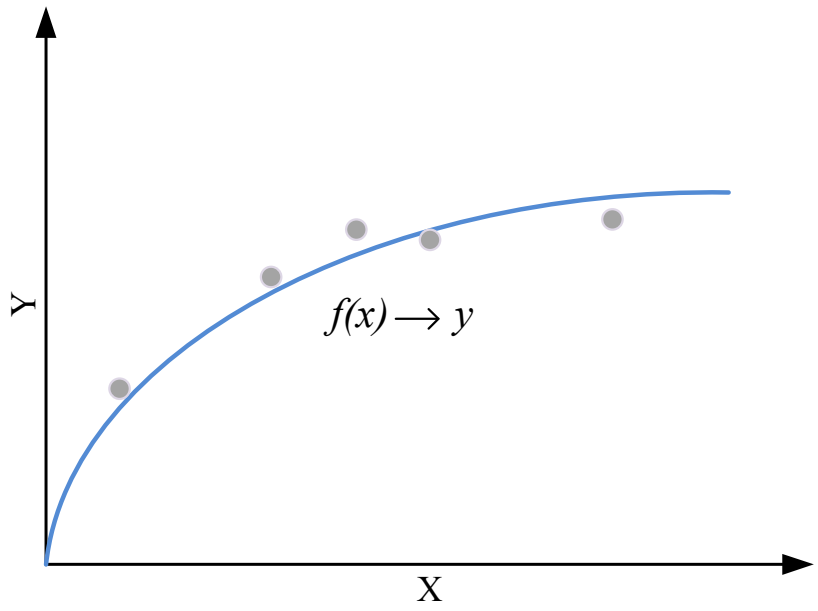


Figure 2.10: A simple curve-fitting example. There are five training examples in this case. A function, f, is trained with the training data, which maps the input x to the output y. The trained function can predict the output of an unseen x.

data defines an abstract, n-dimensional *feature space* where each point in the space is an n-dimensional feature vector. The task of supervised learning is to make use of the training data to learn a function, f, that can accurately predict the corresponding output of an *unseen* feature vector. The function f is called a *predictive* model. Depending on the nature of the outputs, the predictive model can be either a *regression* model for continuous outputs or a *classification* model for discrete outputs.

This section uses a simple curve-fitting example, a regression problem, as presented in figure 2.10 to explain some basic ideas of supervised learning. Suppose we are given five data points shown as circles in this figure. Adhering to supervised learning nomenclature, the set of five known data points is the training data set and each of the five points that comprise the training data is called a training example. Each training example, (x_i, y_i), is defined by a feature vector, x_i, and a desired output, y_i. Learning in this context is understood as discovering the relation between the inputs and the outputs so that the predictive model can be used to make predictions for any new, unseen input features in the problem domain. Once the function, f, is in place, one can use it to make a prediction for a new input. The prediction is the value of the curve that the new input feature vector corresponds to.

In the other subdivision of machine learning, termed *unsupervised learning*, each

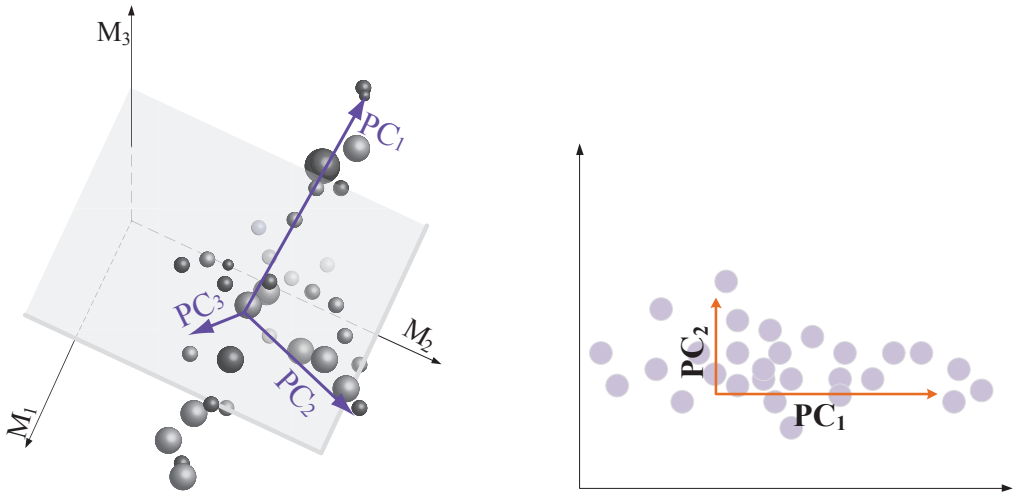
example in the training data is a feature vector merely—there is no output. One form of unsupervised learning is *clustering*, which divides up the feature vectors to a number of subsets. An example of clustering problems would be image compression where the input feature is a RGB value that represents an image pixel. A clustering algorithm can be used to gather pixels with similar colours into one group. By doing this, we can use only a few pixels of each group to represent all the pixels within that group without losing much information.

2.3.2 Principal Component Analysis

Machine learning techniques use features to capture the essential characteristics of a training example. Sometimes the feature space may have a very large number of features; in other words, the dimensionality of the feature space is high. One particular hurdle in dealing with a high-dimensional feature space is that, in many case, it is very difficult to build an accurate predictive model [Bishop, 2006]. Hence, it is of interest to reduce the dimension of the feature space prior to applying machine learning algorithms on the training data.

Principal component analysis (PCA) is a well-established dimension reduction technique [Pearson, 1901; Fodor, 2002]. In essence, PCA uses orthogonal linear transformations to reduce the dimensionality of a set of variables e.g. features. This is achieved by finding a set of *principal component coefficients* to generate a number of uncorrelated combinations (called *principal components*) of the original variables. The number of principal components is less than or equal to the dimensionality of the original data. The principal components are organised in a way that the first component accounts for the greatest variability of the original data; the second one accounts for the second greatest variability of the original data and so on. For many datasets, the first few components usually account for most of the variance of the original data; therefore, the original, high-dimensional data set can be represented with a smaller number of principal components without losing much of variance of the original data.

Figure 2.11 demonstrates the use of PCA to reduce the number of dimensions. The input in this example is a three-dimensional space defined by M_1, M_2 and M_3, as shown in figure 2.11(a). Three components: PC_1, PC_2 and PC_3, which account for the variance of the data, are firstly calculated. One can discover that two components, PC_1 and PC_2 actually contribute most to the variance of the data. Using only these two main components, one can transform the original, three-dimensional space into a new,



(a) (b)

Figure 2.11: Using PCA to reduce dimensionality of a three-dimensional data set. The principal components are computed first (a). Then only the first two principal components are selected and projected into a new two-dimenstional space (b).

two-dimensional coordinate system (as illustrated in figure 2.11(b)) while preserving much of the variance of the original data.

2.3.3 Machine Learning Algorithms

This section gives an overview of the machine learning techniques that are used in this thesis. It first introduces the artificial neural network—a supervised learning technique that can be used for regression. Then, it describes two supervised learning techniques: K-Nearest Neighbour and Support Vector Machines, which can be used for classification. Finally, it describes an unsupervised learning technique called K-Means—a clustering technique that divides up the input data into a number of subsets (called clusters) according to the similarity of features between each data point.

Artificial Neural Network

Artificial Neural Network (ANN) [Bishop., 1996] is a machine learning technique that can model complex problems. A neural network contains an interconnected group of nodes that are connected by weighted links among different layers. The output of a node is determined by the weights and the function associated with the neuron. By changing and updating weights using some adaptive algorithms, the neural network is able to learn from the training data.

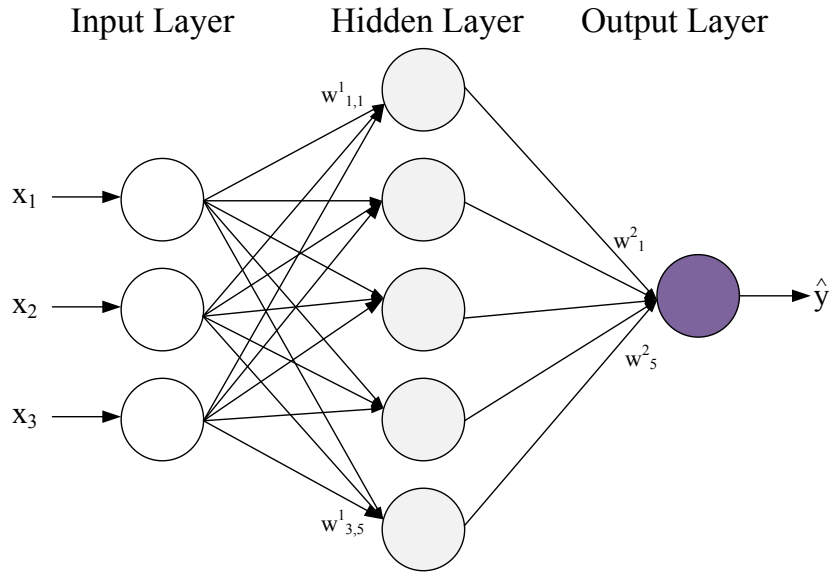


Figure 2.12: A feed-forward network with multiple networks. This network contains three layers: input, hidden and output.

This thesis makes use of a three-layered ANN to the predict the scalability of an OPENMP program in chapter 5. Such a neural network is exemplified in figure 2.12. As can be seen from this figure, data flows forward, from the input nodes, through the hidden nodes, and to the output nodes throughout the network.

The output of a feed-forward network is determined by its network function, $f(x)$, which is defined as a composition of several other functions, $g_i(x)$:

$$f(x) = L(\sum_i w_i \cdot g_i(x)) \tag{2.1}$$

where w_i is a vector of weights of the ith layer and L is an activation function. The activation function can be a *sigmoid* function or a *linear* function. For instance, assuming the neural network shown in figure 2.12 uses a sigmoid function in the hidden layer and a linear function in the output layer, its output, $\hat{y}$, can be formulated as:

$$\hat{y} = \sum_{j=1}^{5} (w_j^2 \cdot sigmoid(\sum_{i=1}^{3} w_{i,j}^1 \cdot x_i)) \tag{2.2}$$

where w^1 and w^2 are the weights associated with the hidden and the output layers, respectively.

When a network is trained using supervised learning, it is given a set of training data. During the training phase, a learning method (e.g. a *back-propagation* algorithm [Russell et al., 1996]) updates the set of weights w of each node according to

some training goals. For example, one could train a network aiming to minimise the mean squared error between the network's output (i.e. the prediction) and the actual output over the training data set.

K-Nearest Neighbour

K-Nearest Neighbour (KNN) [Beyer et al., 1999] is a supervised learning algorithm for classification. In KNN, the class of the input is simply determined by K closest training examples. For a given input, KNN first measures its distance to existing training data in the feature space. Then, the input is classified by a majority vote of its neighbours. As a result, the input is assigned to the class that is most common among its K nearest neighbours. When K is set to 1, the class of the input data is simply determined by its nearest neighbour.

Support Vector Machines for Classification

Support Vector Machines (SVMs) [Boser et al., 1992; Cortes and Vapnik, 1995] are a set of supervised learning methods. When being used for classification, a standard SVM learns from a set of training data where each training example contains an input feature vector and an output that belongs to one of two classes. As shown in figure 2.13, a standard SVM attempts to construct a linear hyper-plane to separate the training data by the maximum margin; in other words, it tries to divide two classes of the data with a gap that is as wide as possible. Once the hyper-plane has been determined, the trained SVM model predicts which class a *new* input feature vector belongs to, through checking which side of the plane the new input lies on.

The training data set for SVM contains some labelled pairs, $(x_i; y_i)$, where x_i and y_i are the feature vector and the class label of the ith training example, respectively. The label indicates which class a training example belongs to. Theoretically, we can find an infinite number of linear hyper-planes in the feature space to separate data, though the separation may not be perfect. The task of a SVM training algorithm is to find a hyper-plane that is likely to give accurate predictions for unseen data. This is achieved by solving the following optimisation problem [Cortes and Vapnik, 1995]:

$$\begin{aligned} &\min_{w,b,\xi} && \frac{1}{2} w^T w + C \sum_{i=1} \xi_i \\ &\text{subject to} && \begin{cases} y_i(w^T \phi(x_i) + b) \geq 1 - \xi_i, \\ \xi_i \geq 0 \end{cases} \end{aligned} \tag{2.3}$$

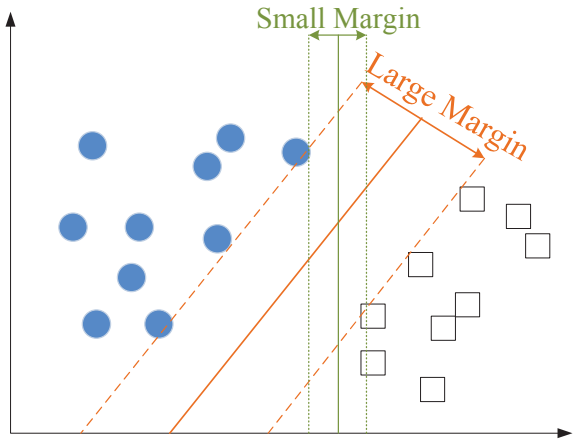


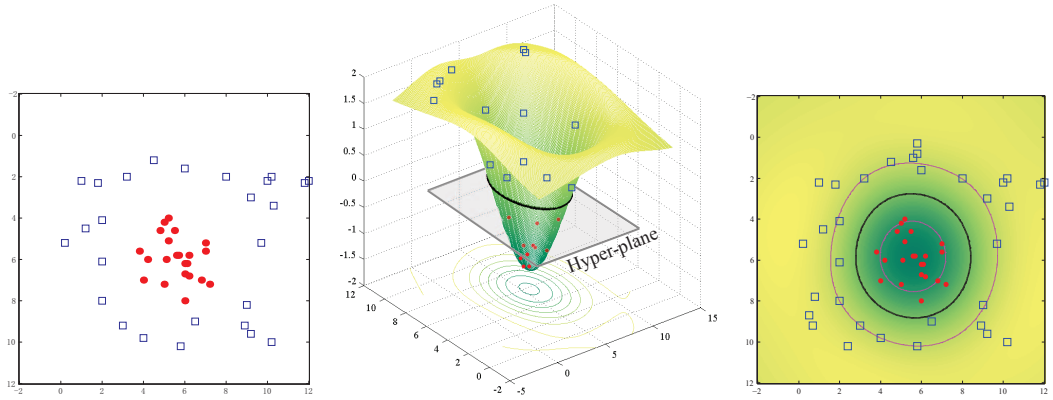
Figure 2.13: Separating data with a hyper-plane with a large margin. A SVM training algorithm attempts to find a linear hyper-plane (i.e. the solid lines in this diagram) to separate the two classes of the training data. The dashed lines drawn parallel to the hyper-plane defines the distance between the hyper-plane and the closest training examples to the line. The distance between the dashed lines is called the *margin*. The points that constrain the width of the margin are called *support vectors*. In this diagram, the hyper-plane with a large margin is better than the other one.

where w is a coefficient vector that is perpendicular to the hyper-planes, C is a cost parameter that indicates how much misclassification can be tolerated on the training data, ξ_i is a slack variable which measures the degree of misclassification of the training example x_i, b is a constant value, and ϕ is a kernel function that is used to transform the feature space of the training data.

Sometimes it is not possible to construct linear hyper-planes to clearly separate training data on the original feature space. In this case, a kernel function can be used to map the input feature vector x_i to a higher dimensional space where it may be easier to find a linear hyper-plane to well separate the data. Among many kernel functions, the radial basis function (RBF) is a reasonable first choice in general [Olson and Delen, 2008], which is defined as:

$$exp(-\gamma \parallel x_i - x_j \parallel^2), \gamma > 0 \tag{2.4}$$

Figure 2.14 illustrates the use of SVM with the RBF kernel to classify two classes of data points. In the original feature space shown in figure 2.14(a), it is impossible to build a linear hyper-plane to separate data points without misclassifying data. By mapping them into a higher-dimensional space shown in figure 2.14(b), we can easily find a linear hyper-plane with a large margin between the two classes of data. This results



(a) The original space (b) A higher dimensional space (c) Non-linear classification

Figure 2.14: Classifying data with SVM using the RBF kernel. It is impossible to construct a linear hyper-plane on the original feature space (a) to separate data points without misclassification. By mapping the original feature space into a higher-dimensional space using the RBF kernel (b), we can find a linear hyper-plane to separate data with a large margin. This leads to a non-linear separation on the original feature space (c).

in a non-linear classification on the original feature space which clearly separates the data into two classes.

The formulation of the standard SVM given in equation 2.4 is based on an assumption that there are only two classes in the training data. The standard SVM can be extended to solve the multi-class problem where the number of classes is greater than two. One way of doing so is to convert the multi-class problem into multiple binary classification problems where each problem yields a binary classifier. For example, for an N-class problem we can build N binary classifiers, one for each class. The ith classifier will be trained to classify whether the data belongs to the ith class or not. When making a prediction for a new input, the ith classifier gives a score to indicate its confidence that the input belongs to the ith class and the classifier with the highest confidence determines the class of the input.

K-Means

As an unsupervised learning technique, *K-Means* uses a measure of similarity between feature vectors in order to split them into closely associated groups [Nilsson, 1996]. The simplest similarity measure involves defining a *distance* between data points in the feature space. The distance measure can be the ordinary Euclidean distance between two data points in the feature space, but other metrics can also be used.

As indicated by its name, K-Means groups given data into K clusters. The objective of K-Means is to minimise the within-cluster *sum of squares* of the cluster members, which is defined as:

$$\sum_{i=1}^{K} \sum_{x_j \in C_i} \| x_j - \mu_i \|^2 \tag{2.5}$$

where x_j is the jth data point in the ith cluster, C_i, and μ_i is the centre (called centroid) of cluster C_i.

Essentially, K-Means is an iterative optimisation algorithm, running as two phases which are repeated until the clusters are converged or the maximum number of iterations is reached. The algorithm begins with a random assignment of K different centroids, then starts its iterative procedure:

Phase 1 For each data point, calculating its distance to each of the K clusters' centroids and then assigning it to the cluster with the closest distance.

Phase 2 For each cluster, adjusting the cluster centroid according to the average distance over all data points that belong to the cluster.

This process iterates until cluster membership (and hence cluster centroids) is stable. As a result, the algorithm chooses a set of centroids and assigns each point to the cluster according to its distance to the cluster centroid. In figure 2.15, a K-Means algorithm is used to gather data points into three clusters. The algorithm groups data points that are close to each other into a cluster.

2.4 Evaluation Methodology

This section describes the methods used throughout the thesis to evaluate the performance of the proposed approaches and the relationship between features and outputs.

2.4.1 Cross-validation

Cross-validation is a statistical method used to evaluate the accuracy of supervised learning techniques [Laan and Dudoit, 2003]. It is a method for assessing how well a supervised learning technique will perform on *new* data that has not been seen in the training phase.

This thesis uses a standard cross-validation technique named *leave-one-out cross-validation* (`LOOCV`) [Bishop, 2006] to evaluate all proposed supervised-learning-based

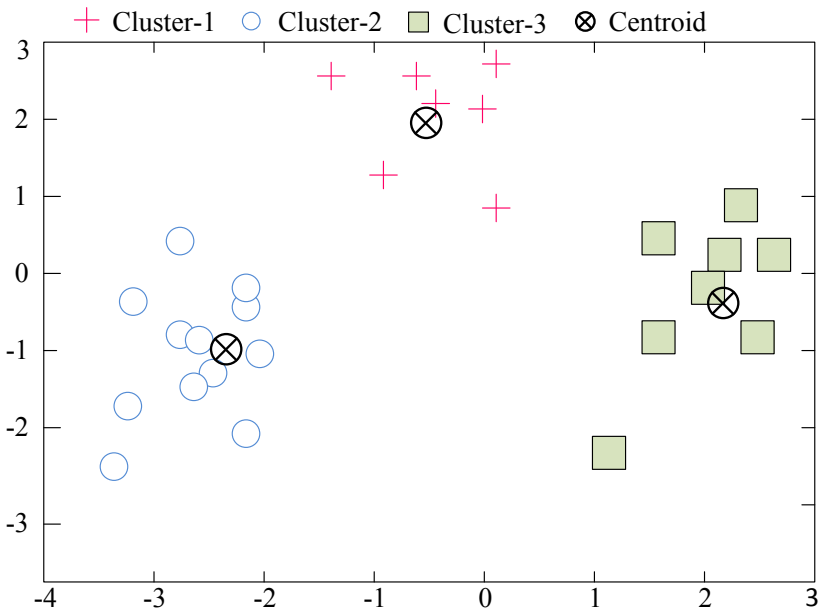


Figure 2.15: Using K-Means to group data points into three clusters. Membership of a cluster is determined by the distance of each data point to the cluster centroid in the feature space.

models. For a given training data set that contains N examples, `LOOCV` first removes one example from the data set, trains a model with the reminding $N-1$ examples, and then uses the trained model to make a prediction for the removed example. This procedure is repeated for each example in turn and the accuracy of the technique is derived by computing the average prediction errors of all examples .

2.4.2 Relative Performance

In practice, it is often desired to evaluate an approach by assessing how close its performance is to the performance achieved by another approach. This thesis uses a metric called "relative performance" to evaluate the closeness of performance between two approaches.

As mentioned before, this thesis aims to reduce the execution time of parallel applications. Accordingly, the "relative performance", p, of approach A to approach B is defined as:

$$p = \frac{T_B}{T_A} \times 100\% \tag{2.6}$$

where T_A and T_B are performance delivered by approaches A and B respectively.

In terms of evaluating the execution time, this is a "larger is better" metric. When p is above 100%, it means approach A achieves better performance when compared to

approach *B*. Whenever it is possible, we often wish to compare one approach to the oracle that always delivers the upper bound performance. In this scenario, the higher the performance relative to the oracle, the better the performance an approach has and 100% means it achieves the best performance.

2.4.3 Correlation Coefficient

The intuition of using supervised learning in solving compilation problems is that similar strategies can be used to optimise similar programs as long as the features can accurately capture similarity between programs. That is to say, it is expected that there is some kind of correlation between features and the expected outputs. It is desired to quantify this relation. One possible way of doing this is using the *correlation coefficient*, a mathematical measure of the correlation between two random variables [Rodgers and Nicewander, 1988]. This coefficient is used for analysing the relationship between the feature vector and the optimisation scheme in chapter 6.

In correlation coefficient, the relation between two random variables: X and Y, is closely related to their *covariance* which is defined as:

$$Cov(X,Y) = E[(X - E[X])(Y - E[Y])] \tag{2.7}$$

where $E[X]$ and $E[Y]$ represents the expected values of X and Y respectively.

Based on their covariance, the coefficient of X and Y is calculated by:

$$\rho_{X,Y} = \frac{Cov(X,Y)}{\sigma_X \cdot \sigma_Y} \tag{2.8}$$

where σ_X and σ_Y are the standard deviations for variables X and Y respectively. The standard deviation σ is defined as:

$$\sigma = \sqrt{(x-\mu)^2} \tag{2.9}$$

The correlation coefficient takes a value between -1 and 1. The greater the magnitude of the correlation coefficient is (ignore the sign), the stronger the correlation between the two variables will be. A correlation coefficient of 1 means that the two variables are perfectly correlated; that is, if one grows so does the other. A correlation coefficient of -1 means that the variables are perfectly inversely correlated; that is, if one grows the other falls. A correlation coefficient of zero means that there is no linear relationship between the two variables.

2.5 Summary

This chapter has introduced multi-core architectures, parallel paradigms, high level parallel programming languages and machine learning techniques that are used in this thesis. It has also introduced the evaluation methodology and metrics which are used to evaluate the performance of the proposed schemes. The next chapter will review the related prior work.

Chapter 3

Related Work

This chapter describes the work that is relevant to this thesis. The first section reviews heuristic- and analytical-based mapping techniques used in static compilers. Section 3.2 investigates iterative compilation techniques that search the best compiler options. Next, section 3.3 examines predictive compilation techniques that directly predict the impact of an optimisation option. Then, section 3.4 discusses runtime adaptation approaches which dynamically schedule parallelism using runtime information. Finally, section 3.5 summarises this chapter.

3.1 Heuristics

The problem of parallelism mapping has been proven to be NP-complete [Indurkhya et al., 1986]. This means that finding an optimal solution by *exhaustively* searching a space of all possible mapping options is time consuming and may be infeasible in reality [Kisuki et al., 1999; Stephenson, 2006]. For this reason, compiler designers have used human intuition and manual trial-and-error experimentation to develop compiler heuristics[1] that find a sub-optimal (instead of optimal) solution at reasonably constrained compile time.

There is usually a strong relationship between heuristics and analytical methods which use a set of formulae to model programs and the target architecture. As illustrated in figure 3.1, a compiler heuristic can rely upon an analytical model to make compilation decisions. The basic idea consists of constructing an analytical-based cost and objective function to steer the process of compilation [Yotov et al., 2003; Stephen-

[1] This thesis refers heuristics as a human-experience-based mechanism that is used for problem solving.

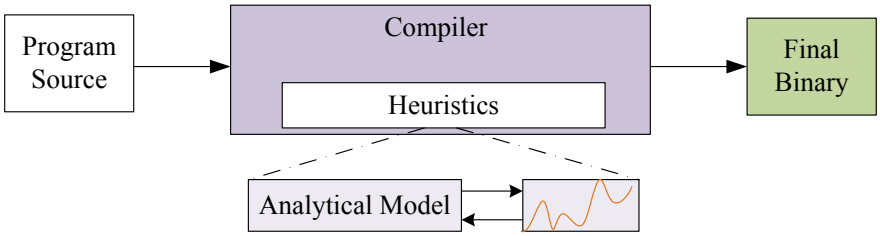


Figure 3.1: Compiler heuristics often use analytical models to evaluate various intermediate forms of the input program during the compilation process so as to generate a final binary.

son, 2006]. Each time the compiler is invoked, an analytical model is used to estimate the benefit of a certain compilation option; based on the estimated information, a search algorithm iteratively traverses the space, searching for a program version that has better estimated performance than the best-so-far.

There has been an extensive body of research on constructing compiler heuristics and analytical models to map parallel programs onto parallel architectures. This section provides a review of some of the proposed techniques on mapping data, task, pipeline and streaming parallelism.

3.1.1 Mapping Data Parallelism

There is much research on applying loop level transformations to achieve data parallelism while trying to manage the overhead associated with the parallel execution. A good survey of loop level transformations can be found in [Wolf and Lam, 1991].

SUIF [Hall et al., 1996] and POLARIS [William et al., 1996] are probably two of the most widely known research parallelising compilers. The two compilers use similar heuristics to decide whether a loop is profitable to be parallelised. Both heuristics count the number of operations and iterations of a loop and only parallelise the loop if the number of operations per iteration is greater than a given threshold. The two compilers also use simple analytical models to estimate the computation and communication of a data parallel loop, through counting the number of operations and array-based memory access, respectively. Based on this statically estimated information, the compilers decide how to distribute the loop iterations across parallel threads.

The IWRAP compilation system uses a set of formulae to estimate the communication cost when partitioning a data parallel loop onto multiprocessors [Balasundaram et al., 1991]. Similarly, in the OPENUH compiler an analytical model is used to estimate the communication to computation ratio in order to find the suitable parallelism

granularity [Liao and Chapman, 2007]. In essence, these approaches are analytical and their success depends on how accurately the analytical model can predict the communication and computation of a given program on the underlying hardware.

Lee et al. propose a scheme to map data parallel applications onto simultaneous multithreading (SMT) multiprocessors [Lee et al., 2010]. Their approach uses static cost models to make mapping decisions at compile time and runtime. At compile time their approach removes parallel loops that only contain a small amount of work. At the runtime their static models use runtime information (e.g. cache miss rates) to determine whether parallel execution of a data parallel loop is profitable and to decide how many threads should be used to execute a loop.

Recently, an attempt to model executions of parallel programs on the CELL multicore processor has been made by Blagojevic et al. [Blagojevic et al., 2008]. The effectiveness of the proposed model is demonstrated by using it to map two computational phylogenetic applications to a CELL platform.

Since an accurate performance model is often crucial to compiler heuristics, a number of analytical models have been proposed to model parallel applications [Valiant, 1990; Fahringer et al., 1992; Anderson and Lam, 1993; Fahringer, 1995; David et al., 1996; Barnes et al., 2008]. The main shortcoming of static performance models is that building such a model requires low-level, detailed knowledge about the underlying hardware [Dubach, 2009]. Therefore, once the target platform has changed, the model must be re-tuned in order to adapt to changes in hardware configurations.

Some implementations allow the programmer to specify different parallel versions in the program source and the compiler tries to pick the best one for a particular runtime environment [Balasundaram et al., 1991; Rauber and Runger, 2000]. However, such approaches place a large burden on developers who have to provide a comprehensive set of program versions for all possible execution environments, if good performance is desired.

Other static mapping techniques have been developed to map data parallel loops, but either with different optimisation objectives or targeting different parallelism levels [Aiken and Nicolau, 1988; Gasperoni et al., 1989; Wolf et al., 1996; Liu et al., 2009]. However, there is not one single heuristic that consistently outperforms others and the best heuristic frequently varies across platforms [Sadayappan and Erçal, 1988; Ruttenberg et al., 1996]. A good review about using heuristics and analytical models to map data parallelism is given in [Gupta et al., 1999].

3.1.2 Mapping Task, Pipeline and Streaming Parallelism

Heuristics and analytical models have also been applied to mapping complicated parallel paradigms, such as task, pipeline and streaming parallelism, because their inherent complex parallel structures often imply a large and complex design space [So et al., 2002; Gordon et al., 2006].

Task Parallelism

One early work on statically mapping task parallelism is made by Sarkar and Hennessy [Sarkar and Hennessy, 1986]. Essential to their compiler-based approach is a static cost function that estimates the communication cost and computation time of a parallel application. To compile the input program, the compiler first expands the program graph to fully expose parallelism. Next, for each parallel task of the expanded program graph, its computation and communication are given by an analytical model. Based on this estimated information, a greedy search algorithm splits up the program graph into regions, through iteratively decreasing the critical path length of the program graph. The partitioned regions are then allocated to threads. The premise of using a greedy algorithm is that by reducing the length of the critical path, the execution time of the input program can be minimised in many cases.

Because static cost models cannot always give accurate estimations, some researchers have proposed the use of profiling information as feedback to build an analytical model that can accurately predict the behaviour of the input program. For example, Subhlok et al. have proposed a feedback-driven mapping approach to compile data parallel applications [Subhlok et al., 1994]. For an input program, this approach first uses profiling information to build a set of program-specific models that can capture the program's behaviour on the target platform. Using the built models, a search heuristic chooses a program version that gives the best estimated performance for the input program.

Pipeline Parallelism

Thies et al. have developed an interactive framework that can help developers manually parallelise a sequential program and then map the parallelised program onto the underlying platform [Thies et al., 2007]. Using the framework, the developer is responsible for providing a partition of the program by marking the pipeline stages of the program source using annotations. Given a marked program, the framework uses dynamic pro-

filing to track the communication patterns between pipeline stages. By analysing the profiling results, the framework presents the communication trace as a stream graph to the programmer—the graph helps him to understand the quality of the partition. Based on the automatically generated stream graph, the programmer can adjust the partition of the pipeline stages until a result that meets some predefined constrains is generated. In essence, this is a semi-automatic mapping approach because the program partitions are manually provided by the programmer and the quality of a partition is determined by the programmer as well.

Raman et al. address the problem of allocating threads to pipeline parallel applications at the basic block level [Raman et al., 2008]. To simplify the problem, their approach only allows one data-parallel stage of the pipeline to be allocated to multiple threads. Their approach uses a greedy-based partitioning heuristic to iteratively merge DOALL (i.e. data parallel) basic blocks until no merging operation is available. As a result, the DOALL basic block with the longest estimated execution time is assigned to multiple threads (so that the execution time of that basic block can be minimised) and the remaining pipeline stages are allocated to a single thread.

Recently, Navarro et al. construct an analytical model to determine parallel configurations, i.e. the number of pipeline stages and communication rates, for pipeline parallel applications [Navarro et al., 2009]. This approach aims to achieve better load balance by applying two techniques: (a) pipeline stage collapsing and (b) dynamic scheduling. The authors have implemented an analytical model based on queuing theory to predict the execution time of a pipeline parallel application. The model is a straightforward approach that takes only the throughput and communication delay of each pipeline stage into account. But, as shown in chapter 6, such a simple approach fails to give robust performance for general applications.

Streaming Parallelism

The ACOTES compiler maps parallelism onto streaming processors by using a static graph partitioning heuristic [Munk et al., 2010]. The heuristic first simulates the input programs to obtain the computation and communication cost. Then, a search algorithm maps the input program onto processors by performing a two-step iterative search. The search algorithm first breaks up the input program graph into several sub-graphs and assigns each sub-graph onto processors; it then refines the partitioned graph by merging tasks to adjust the critical path of the graph. The two phases are repeatedly carried out by the search algorithm until the predefined criteria are met.

The STREAMIT compiler provides two stream graph partitioning heuristics for mapping stream programs onto multi-cores: a dynamic-programming-based and a greedy-based partitioner [Thies et al., 2002; Thies, 2009]. Both are based on static estimations of the computation and the communication costs on a particular platform. The dynamic-programming-based partitioner automatically expands or collapses the program graph as long as the expected load balance improves. By contrast, the greedy-based algorithm does not expanding tasks and instead, it iteratively merges two tasks that have the least estimated work until the number of the generated threads is equal to the number of hardware processors or no tasks can be merged.

The Stream Graph Modulo Scheduling (SGMS) framework orchestrates the parallel executions of streaming applications on the CELL processor [Kudlur and Mahlke, 2008]. SGMS uses integer linear programming (ILP) formulae to transform the input program by aiming to overlap the communication and computation. The ILP formula estimates the computation and communication of a transformation. Given the formulae and some program-dependent parameters, an ILP solver iteratively evaluates the quality of each transformation based on the estimation benefit and cost. Adhering to some constraints, the ILP solver chooses one transformation that best matches the predefined optimisation target. A similar ILP-based approach has also been used to generate partitions for streaming applications but targets GPU architectures [Udupa et al., 2009]. The major drawbacks of the ILP-based approaches are that formulating a ILP model requires expert knowledge about the underlying architecture as well as the problem domain, and a ILP solver typically takes a long time to generate a solution too [Ruttenberg et al., 1996].

More recently, Manley and Gregg improve data locality and data parallelism of stream applications by performing code vectorization [Manley and Gregg, 2010]. Their compiler uses existing vectorization techniques (e.g. code pattern matching) to transform the high-level streaming kernel code into vector instructions. The experimental results show that vectorization can lead to significantly performance improvement for some stream applications on the x86 architecture.

3.1.3 Summary of Heuristic- and Analytic-based Schemes

Fundamentally, heuristics and analytical models are based on the developers' view about the most significant cost of the target platforms and programs. An in-depth knowledge of the hardware and applications is needed in order to achieve satisfac-

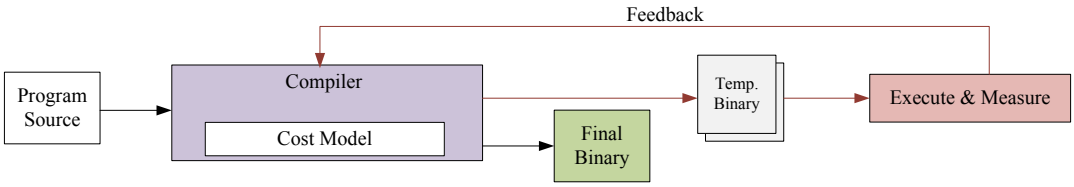


Figure 3.2: Iterative compilation uses profiling information as feedback to generate code.

tory performance. While these techniques can be effective on the designed platforms, their success hinges on the quality of the human-crafted methods. As today's complex systems are difficult to model, it is unlikely that a human-constructed heuristic can adequately describe the cost and benefit of a compilation option [Stephenson, 2006]. Moreover, because of the extensive use of low-level knowledge, porting a heuristic onto a new platform is difficult—which often implies intensive human effort [Hoover and Zadeck, 1996]. The next section describes iterative compilation techniques that can adapt to a platform through compiling and executing many different versions of the input program and then choosing the most efficient one.

3.2 Iterative Compilation

Iterative (or feedback-directed) compilation techniques compile programs based on the empirically measured performance on the target machine [Kisuki et al., 1999]. Figure 3.2 shows a typical iterative compiler, which generates multiple versions of the input program, runs them on the actual hardware (or a simulator), uses the measured performance as feedback and selects the version that has the best measured performance.

Despite some early works in this field which did use exhaustive search to generate solutions [Massalin, 1987; Bernstein et al., 1989], many researchers have realised that to exhaustively search the vast compilation space generally is infeasible [Stephenson, 2006]. Instead of searching the whole compilation space, some other approaches first prune the compilation space and then only perform searching on the most profitable areas in space.

One of the most widely used iterative search techniques is probably the genetic algorithm. For example, the GAPS framework developed by Nisbet uses genetic algorithms to find the best loop transformations in order to minimise the execution time of FORTRAN programs [Nisbet, 1998]. Cooper et al. at Rice University were among the first to use genetic algorithms to reduce the code size on embedded systems [Cooper

et al., 1999]. Their approach is able to improve a compiler that has optimisation phases up to 12, which translates to a search-space size of 10^{12}. Their approach achieves impressive reductions over a standard hand-tuned heuristic. Stephenson et al. use genetic algorithms to tune compiler heuristics, which include hyper-block formation, register allocation and data prefetching [Stephenson et al., 2003]. Under their setting, the intermediate search result is evaluated in a simulator. This approach is able to improve hand-tuned heuristics of the Open Research Compiler [Compiler], demonstrating the efficiency of iterative compilation techniques.

Kisuki et al. propose an interesting approach to select loop tile sizes and unroll factors by aiming to improve data locality and to expose instruction level parallelism [Kisuki et al., 2000]. They implemented several iterative compilation techniques, including genetic algorithms, simulated annealing, grid search, window search and random search, to select loop parameters. Their experimental results show that by dynamically adjusting the search strategies during the process of iterative compilation, an iterative compiler can outperform several well-known static heuristics.

Pouchet et al. design a search algorithm based on genetic programming to rapidly search the loop transformation space that is defined by a polyhedral model [Pouchet et al., 2008]. Fundamentally, their algorithm is an iterative compilation technique that uses the measured execution time as feedback to direct the search. Their technique performs, on average, 2.46x better than a human-generated heuristic. Similar search-based techniques have also been used to schedule instructions [Beaty, 1991] and to find good compiler flag settings [Hoste and Eeckhout, 2008].

Almagor et al. characterise the compilation space of compiler flag settings so as to compare different iterative search techniques [Almagor et al., 2004]. By enumerating different compiler flag settings, they conclude that biased, sampling-based search techniques are able to choose good compiler optimisation options. However, they believe that the long-term goal should be building a model that directly predicts the optimisation goal by using only a few (ideally, zero) times of program execution.

The StreamIt compiler used to have a graph partitioner based on a simulated annealing algorithm—an iterative search technique. This graph partitioner is used to distribute workloads onto the TRIPS multi-core processor [Gordon et al., 2002]. Their implementation supplies the annealing algorithm with three architecture-specific parameters: a cost function, a perturbation function and a set of legal layouts. However, this approach is abandoned in the latest release of the compiler because simulated annealing typically takes a long time to generate a solution.

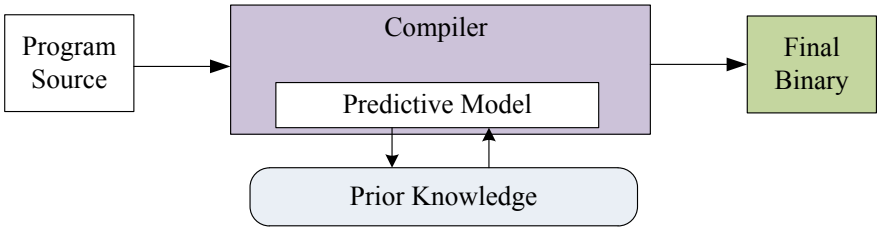


Figure 3.3: Using prior knowledge, a predictive compiler can predict the impact of certain compilation options without actually applying them.

Recently, Dave and Eigenmann propose an auto-tuning framework to select profitable data parallel candidates on multi-core platforms [Dave and Eigenmann, 2009]. This framework generates different combinations of parallel and serial loops, uses profiling runs to measure the whole-program execution time and then selects a version with the best performance.

Essentially, iterative compilation techniques iteratively test multiple code versions of the target program. Evaluating a program version involves compiling and executing the code, which inevitably introduces compilation overhead. Although the compilation space can be pruned, a search is still required for each new program. Predictive schemes presented in the next section, however, use prior knowledge to directly predict the best optimisation setting so that excessive executions of a new program can be avoided.

3.3 Predictive Modelling

In contrast to iterative compilation, predictive modelling techniques predict the impact of optimisations without actually applying them. As shown in figure 3.3, this is achieved by utilising prior knowledge that is obtained *off-line* from executing many training benchmarks with different compiler options. This section examines various predictive compilation approaches that have been developed in the past.

Calder et al. pioneered the use of predictive modelling to solve compilation problems [Calder et al., 1997]. They make use of supervised learning techniques to perform branch predictions. In particular, they use artificial neural networks and decision trees to predict which branch paths are likely to be taken. Using static code features, their approach achieves an accurate prediction rate as 80% compared with 75% which is obtained by the best heuristic at the time.

Moss et al. use supervised learning techniques to learn how to schedule instructions of a ready work list [Moss et al., 1998b]. Using program features, their approach outperforms the scheduling heuristics provided by a production compiler and achieves almost the same level of performance compared to the best instruction scheduler at that time. However, because of the combinatorial number of permutations of instructions, their approach only applies into a fixed, relatively small number of instructions. In fact, the authors recognise that it is difficult to cast the problem into a supervised learning form when the number of instructions is unlimited.

Stephenson and Amarasinghe make use of two supervised classification techniques, K-Nearest Neighbour and Support Vector Machines, to choose a loop unroll factor for a sequential programs [Stephenson and Amarasinghe, 2005]. Their approach only achieves 5% of performance improvement over the default heuristic used by the compiler.

Dubach et al. propose using machine learning techniques to construct portable compilers for sequential programs [Dubach et al., 2009]. Using a K-Nearest Neighbours classifier, their predictive model predicts what compiler settings are good for the input program on a given microarchitecture configuration. Their approach is able to achieve 67% of performance found by using 1,000 evaluations of a single program. With their techniques a new compiler can automatically adapt to the change of microarchitecture configurations.

In recent years several works have made use of regression to optimise parallel applications. For example, Curtis-Maury et al. develop a regression-based model to adjust the OPENMP threads in the context of energy saving [Curtis-Maury et al., 2006]. Their model dynamically samples the execution information of a program and uses hardware performance counters as parameters to make a prediction. Their approach achieves impressive accuracy in ranking configurations. Another example is the Qilin compiler which distributes loop iterations between CPUs and GPUs using regression-based predictions [Luk et al., 2009]. The Qilin compiler builds a regression model by executing the input program with several program inputs. The model is used to estimate the performance of a loop distribution setting for the same program but with different inputs.

This section has introduced several predictive modelling techniques that are used in static compilers. The next section reviews runtime adaptation approaches that make use of runtime information to dynamically map and schedule compiled parallel applications.

3.4 Runtime Adaptation

Runtime adaptation dynamically schedules an application according to runtime information, such as the number of available processors of the runtime environment and the observed execution information of the program. This is different from static compilation techniques which make decisions prior to the beginning of program execution. This section discusses some prior works about using runtime information to dynamically schedule parallel applications on parallel architectures.

Hummel et al. propose a scheme called *factoring* to schedule data parallel loops in a cluster environment [Hummel et al., 1992]. With their technique the iterations of a loop are first divided into a number of batches where different batches have different numbers of iterations. Batches are organised in a way that each batch contains a N times greater number of iterations than its next successive batch. When the loop is being executed, each parallel task will first be given a chunk that has the largest number of iterations and then the remaining chunks are distributed to tasks in a first-come, first-served fashion. By doing this, an early-finished task can perform computation on the next chunk of data quickly without waiting for the unfinished tasks. Essentially, factoring is a trade-off between a fixed-size chunk-partitioning scheme and a dynamic scheduling scheme (where the chunk size is fixed to one). Similar loop scheduling schemes include Guided Self-Scheduling [Polychronopoulos and Kuck, 1987] and Dynamic Self-Scheduling [Fang et al., 1990].

Corbalán et al. have developed a runtime system to dynamically assign OPENMP loops to physical processors [Corbalán et al., 2000]. The system uses an iterative algorithm to allocate processors to a parallel loop. Initially the system allocates a certain number of processors to the loop and measures its execution time via sampling. Using the sampling information, an analytical model is used to estimate the potential performance gain when assigning additional processors to the loop. Using the estimated information of different processor assignments, the runtime allocates a certain number of processors, which gives the best estimated performance, to the target loop.

GALOIS is a programming model that offers a set of C++ templates to specify parallel tasks [Kulkarni et al., 2007]. The GALOIS systems also has a runtime scheduler that adopts a naïve dynamic technique to schedule tasks according to the programmer-specified task priority.

Work-stealing is a technique for scheduling parallel tasks [Blumofe and Leiserson, 1999]. In work-stealing under-utilised worker threads attempt to “steal” jobs from

Table 3.1: Summary of static mapping techniques.

Techniques	Human Effort	Compilation Overhead	Portability
Heuristics	Intensive	**Low**	Poor
Iterative Compilation	**Little**	Expensive	**Good**
Predictive Modelling*	**Little**	**Low**	**Good**

* This thesis makes use of predictive modelling techniques to map data and streaming parallelism. This is contrast with previous works where predictive modelling techniques are mainly used to optimise sequential programs.

other threads, which allows an idle worker thread to perform some of the tasks that are pre-assigned to other threads. Work-stealing has been successfully applied in many parallel programming models such as CILK [Blumofe et al., 1995] and Intel Thread Building Blocks [Blocks].

Runtime adaption is also useful in scheduling stream applications. For example, the FLEXSTREAM system re-maps a compiled stream program "on-the-fly" according to the number of available processors on the underlying platform [Hormati et al., 2009]. Another interesting approach is made by Aleen et al. who combine dynamic profiling information and static estimation to predict the dynamic behaviour of a stream application [Aleen et al., 2010]. The profiling information is used at the runtime execution stage to dynamically adjust the parallel pipeline in order to achieve better runtime load-balance. This approach first uses dynamic information to identify which parts of the program account for the variance of execution time when different input data sets are used. A lightweight, program-specific runtime is built based on the off-line profiling information. When the same program runs with a different input, the runtime scheduler will predict the future execution paths and the program's execution time by looking at the already executed paths. The predicted runtime and execution paths are then used for dynamically assigning pipeline stages to processors.

3.5 Summary

This chapter has presented prior works that are related to parallelism mappings and machine-learning-based compilation. Table 3.1 briefly summarises previous works on the field of static compilation.

Traditionally, many static compilers rely upon either hardwired, platform-specific heuristics which are difficult to be ported onto a new platform, or iterative compilation

schemes that have expensive compilation overhead. By contrast, this thesis uses machine learning techniques to automatically construct predictive models to map parallel applications at compile time. Machine learning makes it possible to construct mapping models that automatically evolve and adapt to architecture change and deliver scalable performance.

Prior works on predictive compilation are mainly restricted to sequential programs that run on single-core processors. In contrast to previous techniques, chapters 4 and 5 will present two novel predictive techniques that map data parallelism across different multi-core platforms.

Furthermore, predictive modelling techniques applied in previous compilers have been mostly limited to relatively straightforward problems where the target to predict is *fixed*, e.g. determining compiler flag settings [Cooper et al., 1999] or loop unroll factors [Stephenson and Amarasinghe, 2005]. This is different from the work presented in chapter 6 where the number of partitioning operations of a stream program is essentially *unbounded*.

Finally, developing an accurate predictive model requires sufficient benchmarks. Prior works have only made use of existing domain-specific benchmarks developed by expert programmers. This is, however, no longer feasible for emerging parallel programming languages where the application code base is relatively small. Chapter 6 will present a synthetic benchmark generator which automatically produces many training benchmarks.

Chapter 4

Mapping Data Parallelism: Identifying Profitable Parallel Candidates

This chapter presents a machine-learning-based technique to identify profitable data parallel candidates of an already parallelised program, which is an important step towards efficient and portable parallelism mappings. It is organised as follows. Section 4.2 motivates the importance of selecting profitable parallel candidates. Section 4.3 explains how a predictive approach is formulated. Then, section 4.4 describes the experimental setup. Finally, section 4.5 discusses the experimental results[1] before section 4.6 concludes the chapter.

4.1 Introduction

One particular problem of compiling a parallel program is that, dependent on the relative costs of communication and computation of the target platform, not all of the parallel candidates (or tasks) are beneficial to the overall parallel execution. In fact, some research has demonstrated that even well-structured parallel applications may run slower than their serial counterparts, if the compiled binary contains unprofitable parallel sections which cannot amortise the overhead (e.g. thread management and communication costs) associated with the parallel execution [Voss and Eigenmann, 1999]. Therefore, a compiler has to determine what parallel candidates are profitable to be parallelised, prior to applying some optimisations. For many cases, the profitability of a parallel candidate has a strong relationship with how it is scheduled [Bull,

[1]The work presented in section 4.5 has been conducted in collaboration with Georgios Tournavitis at the University of Edinburgh. The experiments have made use of the profile-driven auto-paralleliser developed by Tournavitis.

1999]; that is, some candidates are only profitable to be parallelised when the right scheduling policy is used. For this reason, the compiler should take the scheduling policy into account when evaluating the profitability of a parallel candidate.

Despite achieving much progress over the past decades, today's static compilers are still incapable of efficiently detecting the profitability of data parallel candidates [Dave and Eigenmann, 2009]. One important reason is because of the simple models employed by the compilers [Hall et al., 1996; William et al., 1996]. For instance, the Intel ICC compiler [Intel, a], which is probably the best commercial parallelising compiler to date, counts the number of instructions and iterations of a loop and decides whether to parallelise the loop based on a user-defined instruction-iteration-ratio threshold. However, as shown in section 4.2, a simple, human-derived heuristic is not sufficient enough to separate profitable loops from those that are not. Alternatively, auto-tuning frameworks can find the best combination of serial and parallel candidates by executing many different versions of the program on the target machine [Dave and Eigenmann, 2009]. But, due to the nature of these approaches, expensive overhead will be added into the compile time [Kisuki et al., 1999]. Some implementations insert a runtime into the compiled code, which uses dynamically sampled information to decide whether to serialise a data parallel loop or not [Rauchwerger and Padua, 1995; Voss and Eigenmann, 1999]. Nonetheless, the impact of scheduling policies has been neglected by these schemes and their dynamic checking scheme comes at a cost of runtime overhead.

In contrast to previous techniques, this chapter presents an automatic technique to select profitable data parallel candidates at compile time, which can be easily ported onto a new multi-core platform. Rather than relying on hand-crafted techniques that require expert insight into the relative costs of a particular architecture, this chapter describes a machine-learning-based (`ML`-based) technique to decide if a data parallel candidate is profitable to be parallelised and with what scheduling policy the best performance is likely to be achieved.

4.2 Motivation

This section demonstrates that choosing the profitable data parallel candidate[2] has significant impact on performance on multi-core platforms.

[2]Since the experiments in this chapter consider only data parallel loops, we use the terms "data parallel candidate" and "data parallel loop" interchangeably.

```
1 #pragma omp for reduction(+:sum) private(d)
2 for (j=1; j <= lastcol-firstcol-1; j++) {
3     d = x[j] - r[j];
4     sum = sum + d * d;
5 }
```

Figure 4.1: Despite its simplicity, mapping of this parallel loop taken from the NAS *CG* benchmark is non-trivial and the best-performing scheme varies across platforms.

In figure 4.1, a parallel reduction loop originating from the parallel NAS conjugate-gradient (*CG*) benchmark [Bailey et al., 1991] is shown. Despite the simplicity of the code, compilation decisions are non-trivial. For example, parallel execution of this loop is not profitable for the CELL platform due to high communication costs between processing units. In fact, the parallel execution results in a massive slowdown over the sequential version for the CELL for any scheduling policies. On the Intel XEON platform parallelisation can be profitable, but this depends strongly on the specific OPENMP scheduling policy. The best scheme (i.e. STATIC) results in a speedup of 2.3 over the sequential code and performs 115 times better than the worst scheme (i.e. DYNAMIC) that slows the program down to 2% of its original, sequential performance. This example shows that determining whether a parallel candidate is profitable to parallelise has a significant impact on performance and the optimal scheme varies from architecture to architecture.

Separating profitable candidates from those that are not is a challenging task. As we can see from this example, incorrect classification will result in missed opportunities for profitable parallel execution or even in a slowdown due to an excessive synchronisation overhead. Traditional parallelising compilers, such as SUIF-1 [Hall et al., 1996], POLARIS [William et al., 1996] and Intel ICC employ simple heuristics based on the iteration count and the number of operations in the loop body to decide whether or not a particular parallel loop candidate should be executed in parallel. However, as shown in figure 4.2, such a naïve scheme is likely to fail and misclassification will occur frequently. A simple work based scheme would attempt to separate the profitably parallelisable loops by a diagonal line as indicated in the diagram in figure 4.2. Independent of where exactly the line is drawn there will always be loops misclassified and, hence, potential performance benefits wasted. What is needed is a scheme that (a) takes into account a richer set of—possibly dynamic—program features, (b) is capable of non-linear classification, and (c) can be easily adapted to a new platform. The next section describes a predictive modelling approach that has such properties.

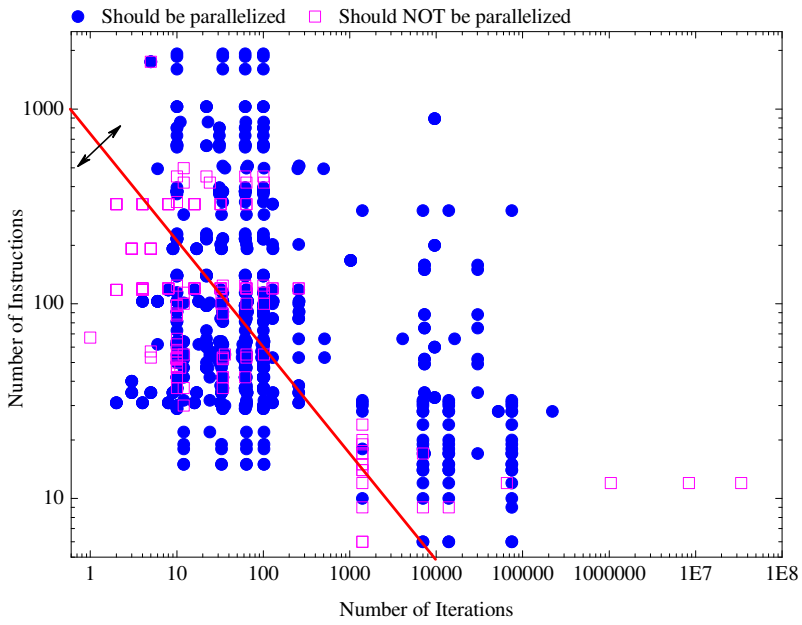


Figure 4.2: This diagram shows the optimal classification (sequential/parallel execution) of all parallel loop candidates considered in the experiments within this chapter for the Intel XEON machine. Linear models and static features such as the iteration count and size of the loop body in terms of Intermediate Representation statements are not suitable for separating profitably parallelisable loops from those that are not.

4.3 Predictive Modelling

This chapter proposes a predictive model based on machine learning. In particular, it uses Support Vector Machines (SVMs) to decide (a) whether or not to parallelise a data parallel candidate and (b) how it should be scheduled. The SVM classifier is used to construct hyper-planes in the multidimensional space of program features to identify profitably parallelisable loops. The classifier implements a multi-class SVM model with a radial basis function (RBF) kernel capable of handling both linear and nonlinear classification problems [Bernhard et al., 1992]. We have made use of the LIBSVM library [Chang and Lin, 2001] to implement the SVM classifier.

In contrast to a simple linear heuristic used by conventional compilers, the proposed SVM model is fundamentally more advanced. Firstly, it uses a larger set of features, including not only static program features such as loop iteration count, but also dynamic program features such as L1 data cache miss rate. Using dynamic program features allow the model to capture the dynamic program's behaviour on the target platform accurately. Secondly, with the RBF kernel, the SVM model is able to perform non-linear classification on the multi-dimensional feature space resulting in better pre-

Table 4.1: Features characterising each parallelisable loop

Static Features	Dynamic Features
IR Instruction Count	L1/L2 DCache Miss Rate
IR Load/Store Count	Instruction Count
IR Branch Count	Branch Miss Prediction Rate
Loop Iteration Count	

diction accuracy than a simple linear classification approach. This is done by using the kernel function to map the original feature space to a high dimensional space in which hyper-planes are produced to separate candidates that are profitable to be parallelised from those are not. Finally, since the model is trained off-line automatically, adapting to a new hardware platform is easy. Whenever targeting a new multi-core platform, the only thing needed to do is to collect training data on that platform. Based on the training data, a learning algorithm can automatically build a model for the target platform.

4.3.1 Program Features

Of crucial importance when developing a ML-based model, is capturing the essential characteristics using features. An overview of these features is given in table 4.1. We have developed a tool for feature extraction. The tool collects program features that sufficiently describe the relevant aspects of a program and present the collected features to the SVM classifier. The static features can be easily derived from a compiler's internal code representation, which characterise the amount of work carried out in the parallel loop. The dynamic features capture the dynamic data access and control flow patterns of the sequential version (i.e. the number of threads is set to one) of the program. They can be obtained using only one profiling run of the target program. In this chapter, this information is supplied by a profile-driven auto-paralleliser (see section 4.4.1).

For each input program, our approach first extracts features from the program and then uses Principal Component Analysis (PCA) (see chapter 2) to reduce the dimensionality of the feature space and presents the reduced features to the SVM model. For the work presented in this chapter, we use PCA to keep 99% of the variance in the extracted features in the training data.

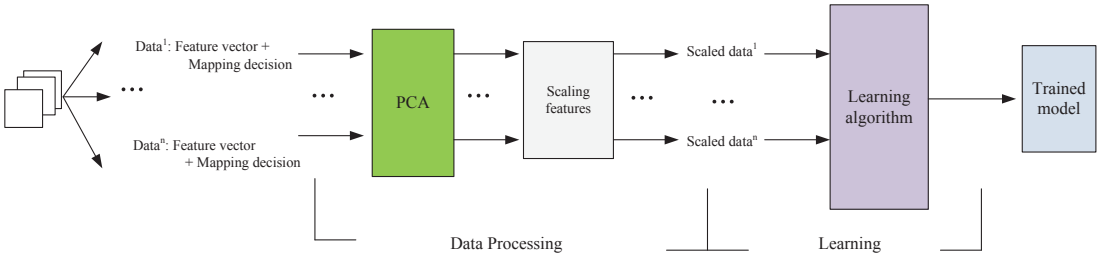


Figure 4.3: Training a SVM model. Each training data is a pair of a feature vector and a mapping decision for a data parallel loop. Firstly, the dimensionality of the feature space is reduced by PCA. Next, the transformed feature vector is scaled into a range between -1 and +1. Then, the normalised feature vectors and mapping decisions are presented to the learning algorithm which finds the SVM model parameters to fit the training data. As a result, a model is built from the training data.

4.3.2 Training the Model

The model is trained *off-line* using supervised learning algorithms whereby the learning algorithm is presented with pairs of program features and desired mapping decisions. These training data are generated through repeated, timed execution of the representative sequential and parallel code with different available scheduling options, and then recording the actual performance on the target platform. In the training process, different SVM parameters are tried by the learning algorithm and the best performing parameters will be selected to train a model from the training data. Once the prediction model has been built using all the available training data, no further learning takes place.

Figure 4.3 shows the process of training a SVM model, which can be broken into two phases: data processing and learning.

During the data processing phase, the training data is pre-processed before being passed to the learning algorithm. For the problem considered in this chapter, each training example is a pair of a feature vector and a mapping decision. In our settings, each feature is a numeric value when it has been extracted. The mapping decisions, i.e. whether a loop should be parallelised or not and the scheduling policy, are actually categorical attributes. In order to present the mapping decisions to the learning algorithm, we convert them into numeric values. For example, we use 1 to represent that a loop is profitable and 0 to represent not. Data processing involves of two steps. (a) Firstly, we find a set of *principal component coefficients* (see chapter 2) from the training data by using PCA. We use the coefficients to transform the original features to a new fea-

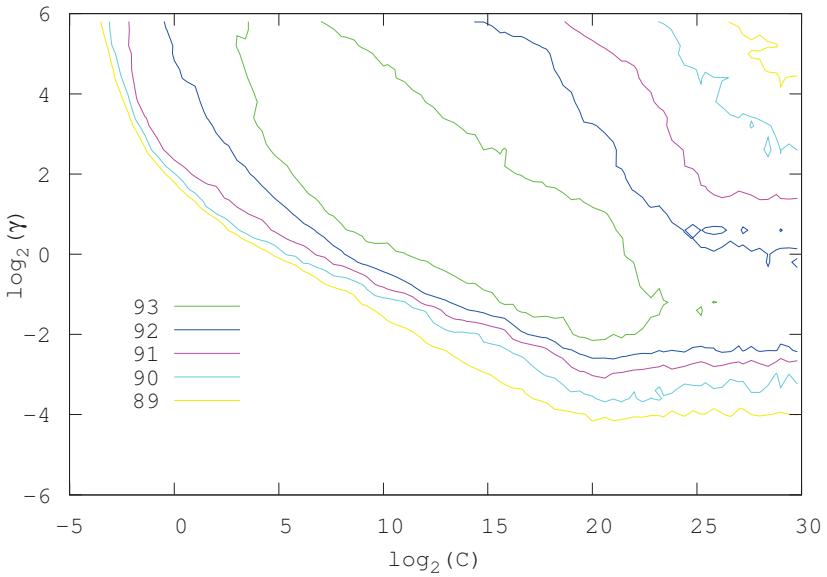


Figure 4.4: Finding suitable SVM model parameters using grid search. In this contour map each curve connects points of a pair of the two parameters that give particular prediction accuracy. The parameter setting that leads to the best prediction accuracy will be chosen as the final model parameters.

ture space where each feature vector in the original feature space is transformed to a new feature vector—each component of the new feature vector is a combination of the components in the original feature vector. For all new feature vectors, we select a fixed, smaller number of components to represent the original feature space. As a result, the dimensionality of the original feature space is reduced. (b) Secondly, because of the nature of the RBF kernel, reduced features must be normalised. This is done by linearly scaling each feature to a range between -1 and $+1$. Scaling features avoids a problem—where features with greater numeric ranges dominate those with smaller ranges—so as to improve the training quality.

In the next phase, the learning stage, a SVM classifier will be built from the training data. The training task here is to construct hyper-planes on the program feature space so that the trained SVM classifier can accurately predict mappings for *unseen* programs. Before constructing any separating hyper-planes, we need to set up the SVM model parameters. There are two parameters to be determined for a SVM model with the RBF kernel: C and γ (see figure **??**). It is not known beforehand what values of these parameters are best for the problem; hence, some kind of parameter search must be performed. We perform the parameter search as follows. Initially, we generate many pairs of the two parameters, $(C;\gamma)$. For each pair of the parameters, we first randomly

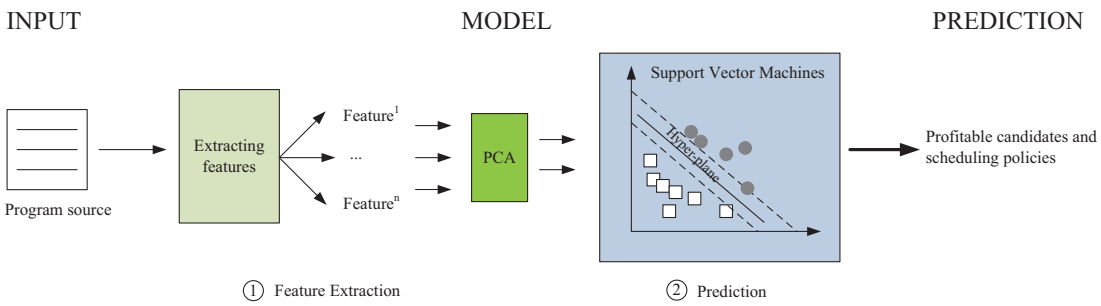


Figure 4.5: Using the trained model. The model first extracts features of each parallel candidate from the input program. Next, PCA is used to reduce dimensionality of the extracted features. Then, the transformed features are presented to the trained SVM model, which selects profitable candidates and predicts scheduling policies for the profitable candidates.

split the *training* data into two different sets – one is used for building hyper-planes with a particular parameter pair and the other is used for evaluating the effectiveness of the parameter pair; then, we train a SVM model using the hyper-plane building data set, evaluate performance of the trained model using the parameter evaluation set, and record the prediction accuracy. For each pair of model parameters, we repeat this procedure (i.e. randomly partitioning the whole training data into two sets) several times and calculate the average prediction accuracy of the SVM models. As a result, we pick the pair of parameters that gives the best average accuracy and use it to train a final model by using the *whole* training data set. Note that during the training process, the training algorithm only performs on the training data set.

The process of parameter search is exemplified in the contour map shown in figure 4.4. In this figure a contour line is a curve that joints different value pairs of the two model parameters, $(C;\gamma)$, which are shown as x and y axis. Different curves represent different prediction accuracy for SVM models with specific parameter settings. As can be seen from this figure there are actually many parameter settings that give high prediction accuracy (i.e. 93% of accuracy on the training data). In other words the accuracy of the model is not very sensitive to particular parameter values.

4.3.3 Deployment

As illustrated in figure 4.5, for a new, previously *unseen* parallel program the following steps need to be carried out in order to evaluate the profitability of data parallel candidates of the program.

Feature Extraction This involves collecting the features shown in table 4.1 from the sequential version of the target program. This information can be collected by using profiling runs of the input program or from a profile-driven auto-paralleliser as we do in this chapter (see section 4.4) .

Prediction For each parallel candidate, the dimensionality of the feature space is reduced. This is achieved by applying the principal component coefficients (which are obtained from the training data by using PCA) on the feature space. The reduced feature space is then presented to the trained SVM predictor which returns a classification indicating whether the parallel candidate is profitable to be parallelised, and what scheduling policy should be used if it is profitable.

4.4 Experimental Setup

This section summarises the experimental methodology and provides details of the multi-core platforms as well as benchmarks used throughout the evaluation in this chapter.

4.4.1 Integrating with a Profile-driven Auto-paralleliser

To demonstrate the effectiveness of the ML-based model, we use it to replace a target-specific mapping heuristic in a state-of-the-art, profile-driven auto-paralleliser that is developed by Tournavitis and Franke [Tournavitis and Franke, 2009]. Figure 4.6 describes the integrated compilation framework. The auto-paralleliser takes a sequential C program as an input and then uses dynamic profiling information (by executing the program with the *smallest* data set) to extract actual control and data dependences of the program. As a result, it annotates all parallelisable loops using OPENMP extensions. Next, the program with parallel annotations is passed to the ML-based model which adds further OPENMP work allocation clauses to the code if the loop is predicted to benefit from parallelisation, or otherwise removes the parallel annotations. Finally, the parallel code is compiled with a native OPENMP compiler for the target platform. Note that the contribution of this chapter is the ML-based mapping model.

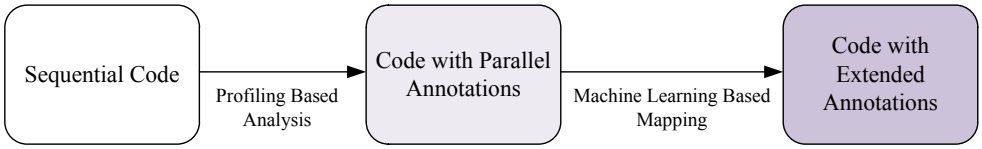


Figure 4.6: A two-stage parallelisation approach combining profile-driven parallelism detection and ML-based mappings to generate OPENMP annotated parallel programs.

Table 4.2: Hardware and software configurations of the two evaluation platforms.

Intel XEON Server	
Hardware	Dual Socket, 2× Intel XEON X5450 @ 3.00GHz 6MB L2-cache shared/2 cores (12MB/chip) 16GB DDR2 SDRAM
O.S	64-bit Scientific Linux with kernel 2.6.9-55 x86_64
Compiler	Intel icc 10.1 -O2 -xT -axT -ipo
CELL Blade Server	
Hardware	Dual Socket, 2× IBM CELL processors @ 3.20GhZ 512KB L2 cache per chip 1GB XDRAM
O.S	Fedora Core 7 with Linux kernel 2.6.22 SMP
Compiler	IBM Xlc single source compiler for CELL v0.9 -O5 -qstrict -qarch=cell -qipa=partition=minute -qipa=overlay

4.4.2 Platforms

In order to evaluate the portability of the proposed approach across different types of multi-cores, experiments are targeted at both shared memory and distributed memory platforms. The first target is a shared memory, homogeneous machine with two quad-core Intel XEON processors, supporting up to 8 threads. The second, in contrast, is a disjoint memory heterogeneous system with two CELL processors, supporting up to 17 OPENMP worker threads. On the CELL platform, one thread runs on the Power Processing Unit (PPU), the remaining 16 on each of the Synergistic Processing Units (SPUs). A brief overview of each platform's characteristics including the operating system and the baseline compiler is given in table 4.2.

Table 4.3: Benchmark applications used

Application	Suite	Inputs on XEON	Inputs on CELL
BT	NPB2.3-OMP-C	S, W, A, B	NA
CG	NPB2.3-OMP-C	S, W, A, B	S, W, A
EP	NPB2.3-OMP-C	S, W, A, B	S, W, A
FT	NPB2.3-OMP-C	S, W, A, B	S, W, A
IS	NPB2.3-OMP-C	S, W, A, B	S, W, A
MG	NPB2.3-OMP-C	S, W, A, B	S, W, A
SP	NPB2.3-OMP-C	S, W, A, B	S, W, A
LU	NPB2.3-OMP-C	S, W, A, B	S, W, A

4.4.3 Benchmarks

The proposed model is evaluated by mapping OPENMP programs onto the two multi-core platforms. The NAS parallel benchmark (NPB), which has both sequential and manually parallelised OPENMP versions available, was used in the experiments. This has enabled us to directly compare the ML-based mappings against mappings performed by independent expert programmers. More specifically, the NPB sequential v.2.3 and NPB OPENMP v.2.3 benchmarks are used in the experiments. Each program has been executed using multiple different input data sets (shown in table 4.3); however, for feature extraction, only the *smallest* available data set is used.

Data Sets Each program was executed using multiple inputs (as described in table 4.3). Programs were run with the S, W, A and B classes on the XEON platform, and with S, W, A on the CELL platform. On the CELL platform, class B could not be executed due to the limited local memory of the SPU.

4.4.4 Candidate Mapping Approaches

Throughout the experimental results section we refer to three different mapping approaches: *manual*, *icc* and *profile-driven auto-paralleliser*.

Manual The results here correspond to compiling and executing the 'C' version NAS parallel benchmark. This is a well-studied parallel benchmark suite which has been tested and optimised on an extensive set of parallel architectures.

ICC The Intel ICC compiler has an advanced auto-parallelisation facility and is probably the most state-of-the-art commercially available parallelising tool. Clearly, it is only available on the Intel platform, so results are only presented for this platform. It does not generate OPENMP as intermediate code, instead going directly to binary. It is possible to externally parameterise its profitability-evaluation decisions based on parallelisation profitability. The experiments consider two parallelisation settings:

(A) ICC (default): Intel ICC with the default profitability threshold for deciding if a loop should be parallelised.

(B) ICC (runtime): Intel ICC with runtime checking of the profitability threshold.

These compiler settings are applied to the sequential version of the corresponding NAS benchmark.

Profile-driven Auto-paralleliser Finally we present the results of the profile-driven auto-paralleliser, which uses dynamic profiling information to discover data parallelism and then maps discovered parallelism onto different platforms. For each program, the process of using dynamic information to discover parallelism and obtaining dynamic program features only needed to carry out once and the profiling is performed on the smallest available data set. The discovered parallelism is annotated with OPENMP clauses in the program source—the generated OPENMP code contains all parallel candidates found by the auto-paralleliser. Next, a mapping model is used to select profitable candidates from the generated code. Finally, the mapped OPENMP code is compiled by the backend compilers on the XEON and CELL platforms which are described in table 4.2. For the purpose of the experiments, two different mapping strategies are considered:

(A) Profiling + Heuristic: Profile-driven auto-paralleliser with a fixed per architecture mapping heuristic.

(B) Profiling + Machine Learning: Profile-driven approach with the `ML`-based cross-platform mapping technique.

Each of these approaches was applied to the NAS benchmark suite and evaluated on the XEON and the CELL platforms. For native code generation, all programs with both the sequential and the parallel OpenMP versions have been compiled using the Intel ICC and IBM XLC compilers on the Intel Xeon and IBM Cell platforms, respectively.

4.4.5 Evaluation Methodology

The ML-based mapping model is evaluated by using leave-one-out-cross-validation, a standard evaluation method described in chapter 2. This ensures the target program is not seen by the ML-based model during the training phase and as a result, the ML-based model always makes predictions for an unseen program in the experiments.

Measurement

Rather than presenting only speedup results compared to the sequential version, we compare our approach with the *well-established, hand-parallelised* version of the NAS parallel benchmark suit. These parallel applications have been evaluated with parallel compilers, including the Intel ICC [Xinmin et al., 2003] and the IBM XLC OPENMP compiler for CELL [Eichenberger et al., 2005] and should therefore provide a fair comparison.

In the experiments, each benchmark was executed 10 times and the minimum execution time is selected. As we have found, there was little variance in the execution time and on average, it was less than 0.5%. Furthermore, in contrast to some prior research on the CELL platform that evaluates parallel performance against a single SPU thread [Eichenberger et al., 2005; Gonzàlez et al., 2008] or a single OPENMP thread [Lee et al., 2008], the performance of all approaches is compared with a sequential execution of the program on the general-purpose PPU where the strongest baseline is available.

4.5 Experimental Results

This section firstly evaluates the overall performance of the profile-driven auto-paralleliser but with a ML-based mapping model on both platforms, through comparing the “profiling + machine learning” approach against hand-parallelised programs. Then, the ML-based mapping scheme is compared with the manual and the fixed mapping heuristics. Finally, the scalability of each benchmark is presented on both platforms.

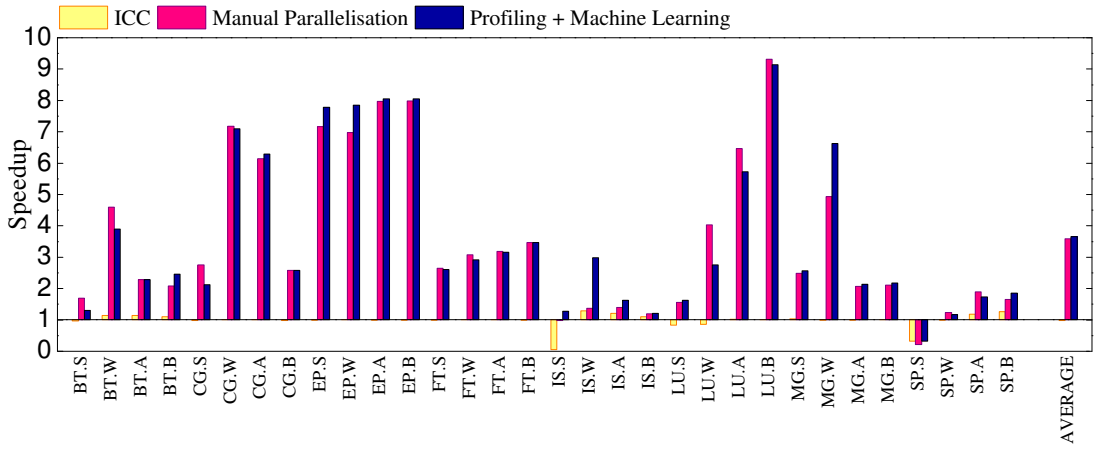


Figure 4.7: Speedup over sequential codes achieved by ICC, manual, and "profiling + machine learning" approaches for the XEON platform.

4.5.1 Profiling + Machine Learning

Intel XEON Figure 4.7 summarises the performance results on the Intel XEON platform. The most striking result is that the Intel auto-parallelising ICC compiler fails to exploit any usable levels of parallelism across the whole range of benchmarks and data set sizes. In fact, auto-parallelisation results in a slow-down of the *BT* and *LU* benchmarks for the smallest and for most data set sizes, respectively. ICC gains a modest speedup only for the larger data sets of the *IS* and *SP* benchmarks. The reason for this disappointing performance of the Intel ICC compiler is that the compiler's mapping heuristic typically chooses to parallelise the inner-most loop where significant fork/join overhead negates the potential benefit from parallelisation. The manually parallelised OPENMP programs achieve an average speedup of 3.5 across the benchmarks and data sets. In the case of *EP*, a speedup of 8 was achieved for large data sets. This is not surprising since this is an embarrassingly parallel program. More surprisingly, *LU* was able to achieve super-linear speedup (9x) due to improved caching [Grant and Afsahi, 2007]. Some programs (i.e. *BT*, *MG* and *CG*) exhibit lower speedups with larger data sets (A and B in comparison to W) on the XEON machine. This is a well-known and documented scalability issue of these specific benchmarks [Grant and Afsahi, 2007]. For most NAS benchmarks the "profiling + machine learning" approach achieves performance levels close to those of the manually parallelised and mapped versions, and sometimes outperforms them (*EP*, *IS* and *MG*). The performance improvement can be attributed to two important factors. Firstly, the auto-paralleliser tends to parallelise the outer loops whereas the hand-parallelised codes contain parallel inner loops. Sec-

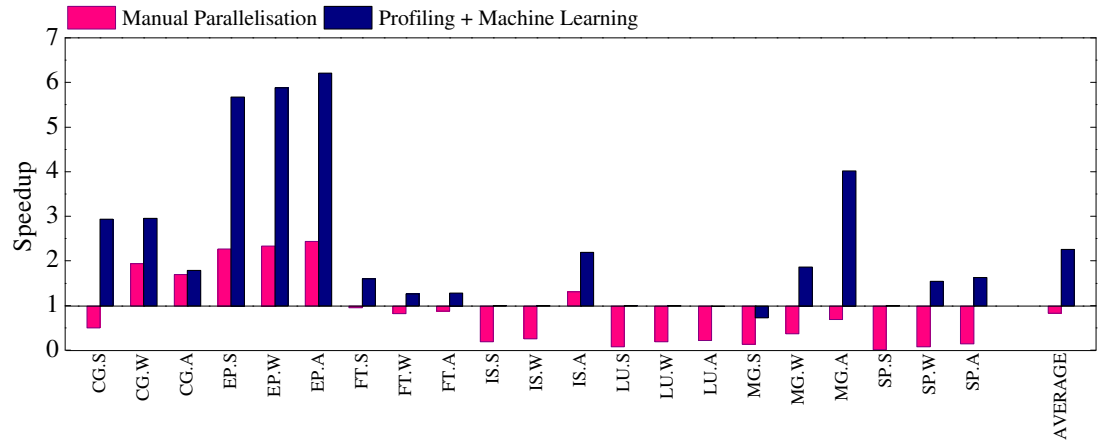


Figure 4.8: Speedup over sequential code achieved by manual parallelisation and profile-driven parallelisation for the dual CELL platform.

ondly, the ML-based model is more accurate in eliminating non-profitable loops from parallelisation.

Overall, the results demonstrate that with the help of the ML-based scheme, the profile-driven auto-paralleliser significantly improves on the state-of-the-art Intel ICC compiler. In fact, the integrated approach delivers performance levels close to or exceeding those of manually parallelised codes and, on average, it achieves 6% of performance improvement over the manually parallelised code.

IBM CELL Figure 4.8 shows the performance resulting from manual and "profiling + machine learning" approaches for the dual-CELL platform. Unlike the XEON platform, the manually parallelised OPENMP programs do not deliver a high performance on the CELL platform. On average, the manual implementation results in an overall slowdown. For some programs such as *CG* and *EP* small performance gains could be observed, however, for most other programs the performance degradation is disappointing. Given that these are hand-parallelised programs this is perhaps surprising and there are essentially two reasons why the CELL's performance potential could not be exploited. Firstly, it is clear that the OPENMP implementations have not been specially mapped onto the CELL platform. The programmer has not considered the communication costs for a distributed memory machine. Secondly, in the absence of specific scheduling directives the OPENMP runtime library resorts to its default behaviour, which leads to poor overall performance. Given that the manually parallelised programs deliver high performance on the XEON platform, the results for the CELL demonstrate that effectively expressing parallelism in *isolation* is not sufficient, but

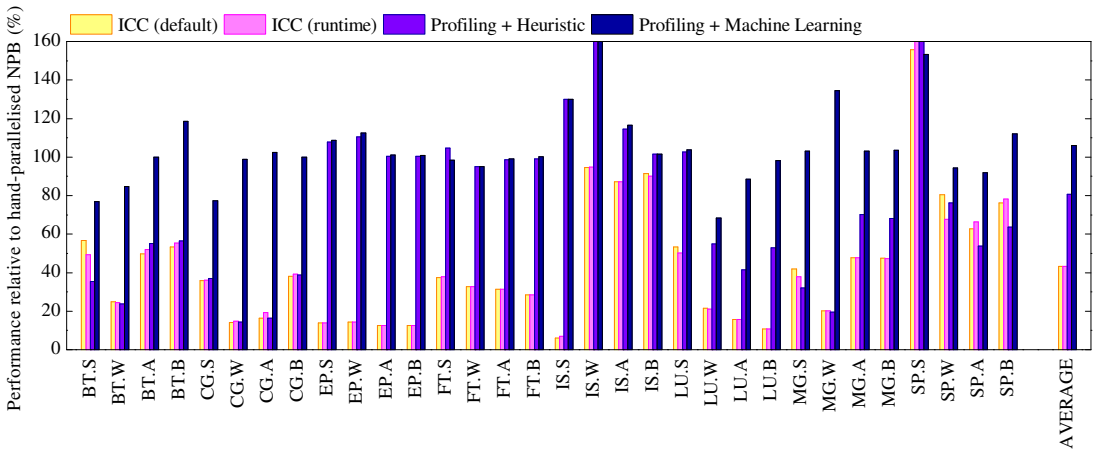


Figure 4.9: Performance relative to the performance of the hand-parallelised version code on the Intel 8-core platform for ICC, profiling + heuristics and profiling + machine learning versions. The profiling + ML-based approach achieves the best performance.

mapping must be regarded as equally important. In contrast to the "default" manual parallelisation scheme, the ML-based model is able to successfully exploit significant levels of parallelism, resulting in average speedup of 2.0 over the sequential code and up to 6.2 for individual programs (*EP*). This success can be attributed to the improved mapping of parallelism resulting from the ML-based mapping approach.

4.5.2 Parallelism Mapping

This section examines the effectiveness of three mapping schemes, i.e. manual, heuristic with static features and machine learning, across the two platforms.

Intel XEON Figure 4.9 compares the performance of ICC, "profiling + heuristic" and "profiling + machine learning" approaches to that of the hand-parallelised OPENMP programs. In the case of ICC this figure shows the performance of two different mapping approaches. By default, ICC employs a compile-time profitability check while the second approach performs a runtime check using a dynamic profitability threshold. For some cases (*BT.B* and *SP.B*) the runtime checks provide a marginal improvement over the static mapping scheme while the static scheme is better for *IS.B*. Overall, both schemes are equally poor and deliver less than half of the speedup levels of the hand-parallelised benchmarks. The disappointing performance appears to be largely due to non-optimal mapping decisions, i.e. to parallelise unprofitable inner loops rather than outer ones.

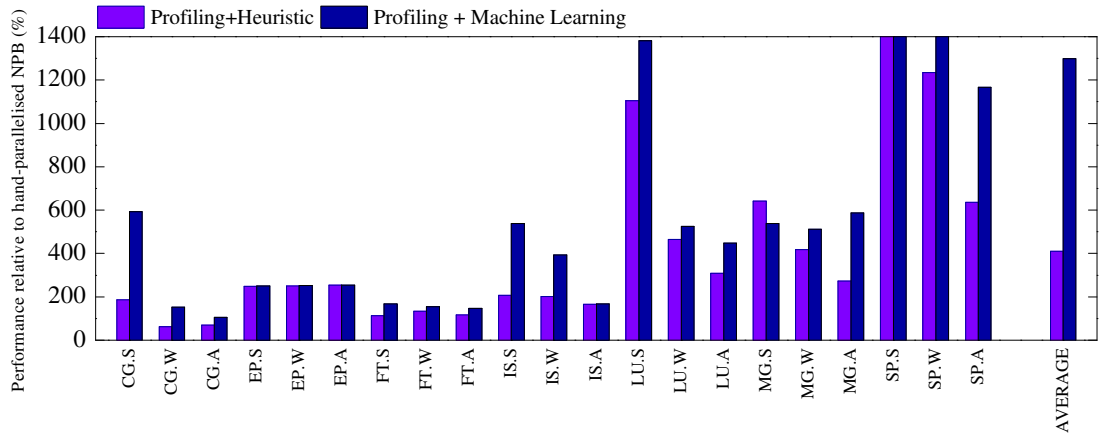


Figure 4.10: Speedup over the hand-parallelised programs on the CELL platform. The ML-based mapping significantly outperforms the fixed mapping heuristic.

The same figure compares the ML-based mapping approach against a scheme which uses the same profiling information, but employs a fixed, work-based heuristic similar to the one implemented in the SUIF-1 parallelising compiler (see also figure 4.2). This heuristic considers the product of the iteration count and the number of instructions contained in the loop body and uses a threshold to decide the profitability. In many cases, the machine learning approach is able to outperform the hand-parallelised codes and, on average, achieves 6% of performance improvement over them. The static heuristic performs poorly and is unable to obtain more than 80% of the performance of the hand-parallelised code. This translates into an average speedup of 2.5 rather than 3.7. The main reason for this performance loss is that the default scheme uses only static code features and a linear work model is unable to accurately determine whether a loop should be parallelised or not.

IBM CELL Figure 4.10 shows the speedup of the ML-based mapping approach over hand-parallelised codes on the CELL platform. As before, this diagram compares the ML-based model against a scheme which uses the same information from the profiling-based auto-paralleliser, but employs a fixed mapping heuristic. The manually parallelised OPENMP programs are not specifically "tuned" for the CELL platform and perform poorly. As a consequence, the ML-based mappings show high performance gains over this baseline, in particular, for the small input data sets. Still, the ML-based mapping scheme outperforms the fixed heuristic counterpart by far and, on average, results in a speedup of 2.7 over the fixed mapping heuristic which translates to a speedup of 3.7 over the hand-parallelised OPENMP programs across all data sets.

Summary of ML-based Mappings

The ML-based parallelism mapping scheme significantly outperforms fixed, work-based mapping heuristics. It improves the original heuristic used by the profile-driven auto-paralleliser by large margins, resulting in close performance compared to hand-parallelised codes on the Intel XEON platform and in some cases outperforming it. Fixed heuristics are not strong enough to separate profitably parallelisable loops from those that are not and perform poorly. Typically, static mapping heuristics result in performance of less than 60% of the machine learning approach. This is because the default schemes used in both the profile-driven auto-paralleliser and the Intel ICC compiler are unable to accurately determine whether a loop should be parallelised or not. The situation is exacerbated on the CELL platform where accurate profitability evaluations are key enablers to high performance. Existing ("generic") manually parallelised OPENMP code fails to deliver any reasonable performance and heuristics—even if based on a hardwired mapping heuristic tuned on the target platform—are unable to match the performance of the ML-based scheme.

4.5.3 Mapping Decisions

This section presents decisions made by our ML-based model on the XEON platform with respect to two metrics: "the average execution time per loop iteration" and "the execution time per loop to the whole program's execution time", which provide further details about our approach.

Figure 4.11 shows the decisions made by the ML-based model with respect to the mean execution time per loop iteration. The loop execution time is measured by the auto-paralleliser during the profiling runs of all benchmarks. In the figure, the x-axis shows the number of parallelisable loops considered in the experiments and the y-axis shows the mean execution time per loop iteration at a *log* scale. In this figure, the larger the radius a circle has, the longer the mean execution time a loop has. In some cases, a long run loop has many loop iterations; therefore, its execution time per loop iteration can be relative shorter than other loops. As can be seen from this diagram, our ML-based model does not only rely on the mean execution time when determining the profitability of a parallel loop. In fact, some loops, which have long sequential execution time per iteration, are identified to be unprofitable. This is because some other characteristics of these loops, such as the number of memory operations

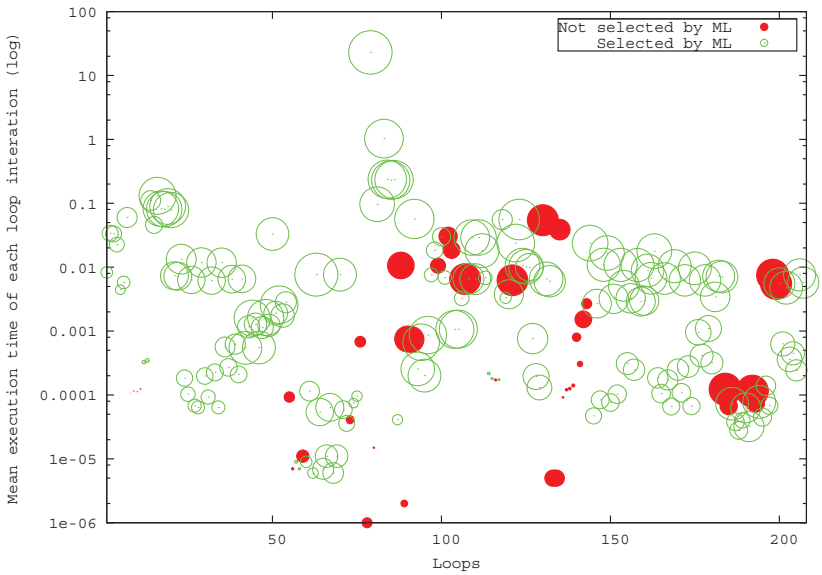


Figure 4.11: Mapping decisions made by the ML-based model with respect to mean execution time per loop iteration on the XEON platform. The larger the radius a circle has, the longer the execution time a loop has. This diagram shows that the mean execution time per iteration is not the only metric considered by our ML-based model.

or execution branches, determine that they are not profitable to be parallelised on the XEON platform.

Other parallelisation techniques consider the fraction of sequential execution time that is spent on a loop, and only parallelise loops that contribute most to the whole-program execution time [Zhong et al., 2008]. However, as shown in figure 4.12, our ML-based approach decides not to parallelise some performance-critical loops, though they account more for the sequential program execution time than some others that are identified to be profitable. That is to say, our ML-based model also considers other factors when determining the profitability of a parallel loop.

These two figures demonstrate that our ML-based model does not rely upon simple metrics to make predictions. They to some extent explain why the ML-based model achieves better performance than a platform-specific, work-based heuristic—the ML-based model considers a richer set of program features.

4.5.4 Scalability

It is important to examine how performance scales with the number of processors to see to what extent parallelism may be exploited beyond the platforms we consider in this chapter. Figure 4.13 shows the scalability of benchmarks mapped by our ML-based

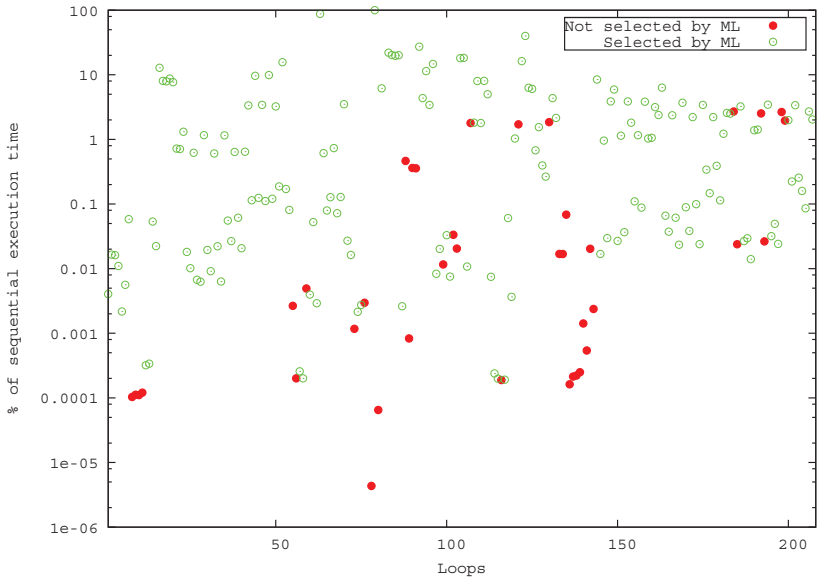
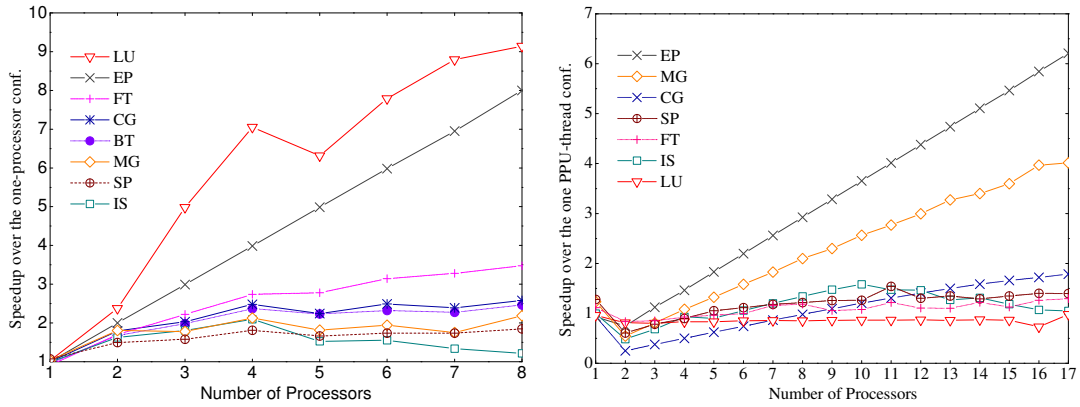


Figure 4.12: Mapping decisions made by the ML-based model on the XEON platform, with respect to the percentage of execution time per loop that is relative to the whole program's execution time. This diagram shows that our ML-based approach may identify some loops as unprofitable candidates, though they may contribute much to the whole program execution time.

model on the two platforms. On the XEON platform, for some programs, adding additional processors generally leads to increased performance. For example, the *LU* and *EP* benchmarks scale well as the number of processors increases (see figure 4.13(a)). In fact, a super-linear speedup due to more cache memory in total can be observed for the *LU* benchmark. The scalability of other applications is more limited and often saturation effects occur for four or more processors on the XEON platform. Actually, we are not the first to discover the scalability issue of the NAS parallel benchmark suite, prior research on the XEON platform has shown similar problems [Grant and Afsahi, 2007]. Figure 4.13(b) shows the scalability of the benchmarks on the CELL platform. According to this figure, the application performance drops for the step from one to two processors on this platform. This is due to the fact that we use the generally more powerful PPU to measure single processor performance, but then use the multiple SPUs and one PPU for parallel performance measurements. The diagram reveals that in the best case it takes around three SPUs to achieve a comparable performance of using one PPU. Some of the more scalable benchmarks such as *EP* and *MG* follow a linear trend as while for some programs, adding additional processors generally leads to increased performance because the compute to communication ratio does not justify using more parallel resources. One interesting observation here is that the number of processors



(a) Intel XEON Platform (b) IBM CELL platform

Figure 4.13: Scalability of the benchmarks that are mapped by our ML-based model on the XEON (a) and the CELL (b) platforms. The two diagrams show the scalability using the largest input data set.

allocated to a program has a great impact on performance and should not be neglected. Chapter 5 will address this problem.

4.6 Conclusions

This chapter has presented a machine-learning-based approach to decide whether a parallel loop is *profitable* to be parallelised on a particular platform by taking the scheduling policy into account. The model is based on a support vector machine classifier that is built off-line and makes predictions for any new input. Compared with traditionally fixed-target mapping techniques, the ML-based mapping model provides more scope for adaptation to different architectures.

Experimental results obtained on two complex multi-core platforms (Intel XEON and IBM CELL) with the NAS parallel benchmark suite confirm that the ML-based mapping scheme is more effective and portable than existing hardwired mapping heuristics. On average, it achieves speedups of 1.31 and 2.7 over a hardwired, target-specific mapping heuristic on Intel XEON and IBM CELL platforms, respectively.

The next chapter will consider how many threads should be allocated to a profitable parallel candidate and how the candidate should be scheduled with a given number of threads.

Chapter 5

Mapping Data Parallelism: Determining the Best Number of Threads

The last chapter has presented a predictive model that decides whether a data parallel candidate is profitable to be parallelised. Once profitable candidates have been identified, the next step is to determine how to allocate hardware resource, i.e. how many threads should be used to execute a parallel candidate. In this chapter, we propose two compiler-based, automatic and portable models to predict how many threads should be allocated to a given data parallel candidate. The two schemes make predictions for any *unseen* programs across different input data sets.

The rest of this chapter is structured as follows: section 5.2 demonstrates that it is a non-trivial task to predict the right number of threads on a particular platform. Section 5.3 describes how the machine-learning-based models are constructed, trained and deployed. Then, sections 5.4 and 5.5 discuss the experimental methodology and results. Finally, section 5.7 summarises the contributions of this chapter.

5.1 Introduction

The number of threads used to execute a data parallel candidate (which is referred to as a data parallel loop in this chapter) has great impact on its performance [Earl et al., 2005]. Although developers can manually tune a particular application with trial-and-error experimentation and then embed the "best-found" numbers of threads for all the parallel loops in the code, this is the most time-consuming approach. Even

if developers were able to specify the optimal numbers of threads for a single program with a particular program input, determining the optimal thread configurations for a diverse set of inputs is a formidable task. Therefore, it is desired to have a technique that can *automatically* determine the best thread configurations for a given data parallel loop across multiple program inputs.

Unfortunately, implementing such a scheme by human expertise alone is difficult because of the complex interactions between parallel programs and architectures. This is probably why many OPENMP runtime schedulers have to conservatively allocate all processors (i.e. threads) to a parallelisable loop by default [Kusano et al., 2000; Intel, a; Eichenberger et al., 2005; GCC]. However, as shown in section 5.2, using all processors to execute a parallel loop may result in poor performance in practice. Alternatively, some implementations use heuristics to decide the number of threads for a loop with a given input, by observing the runtime behaviour of the target program [Corbalán et al., 2000; Duran et al., 2005; Kulkarni et al., 2007]. Though these approaches can be efficient on the designed platform, their performance depends on the quality of the platform-specific heuristics and as a result, once the underlying architecture has changed, the model must be re-tuned by hand. Manually tuning heuristics for today's complex processors, however, is no longer feasible because of rapidly evolving architectures. What is needed, indeed, are techniques that (a) can make predictions across different program inputs and (b) can automatically adapt to diverse platforms.

This chapter presents an automatic and portable approach to predict the best number of threads for any given data parallel loop across different input data sets. Using machine learning, this approach is able to accurately predict the program's behaviours across input data sets at a fine-grain level, allowing a compiler to select the best thread configuration at compile time.

5.2 Motivation

In this section, we demonstrate that allocating the right number of threads to a parallel loop has great impact on its performance and the best number of threads varies across architectures.

In figure 5.1, a critical loop in the NAS *FT* benchmark is shown. Despite the simplicity of the code the parallelism mapping decisions are certainly non-trivial. Similarly to chapter 4, we consider the performance of this loop on two different multi-core architectures: a homogeneous platform with two 4-core Intel XEON processors (8

```
1  #pragma omp for
2  for (i = 0; i < dims[2][0]; i++) {
3      ii =  (i+1+xstart[2]-2+NX/2)%NX - NX/2;
4      ii2 = ii*ii;
5      for (j = 0; j < dims[2][1]; j++) {
6          jj = (j+1+ystart[2]-2+NY/2)%NY - NY/2;
7          ij2 = jj*jj+ii2;
8          for (k = 0; k < dims[2][2]; k++) {
9              kk = (k+1+zstart[2]-2+NZ/2)%NZ - NZ/2;
10             indexmap[k][j][i] = kk*kk+ij2;
11         }
12     }
13 }
```

Figure 5.1: A simple parallel loop from the NAS *FT* benchmark. Determining the best number of threads for this loop is non-trivial.

cores in total) and a heterogeneous platform with two IBM CELL processors (18 cores in total). On the XEON platform, the best parallel configuration is to use all 8 processors with the CYCLIC scheduling policy. The question is whether this is also a good parallel configuration for the same loop on the CELL platform? Consider figure 5.2 where the x-axis shows the number of work threads used for this loop on the CELL platform, and the y-axis shows the speedup obtained for it. This graph shows that selecting the right number of threads is important, if we want to obtain any speedup. There is no performance improvement available by using more than one worker thread—where the additional threads are executed by the Synergistic Processing Units (SPUs) of the CELL processor. This is because executing this loop with any SPUs will introduce intra-core communication overhead that cannot be amortised by the parallel execution. In fact, the best strategy on this platform is to spawn only one worker thread on the Power Processing Unit (PPU) which has a relatively low communication cost when accessing the computation data of the loop. On the CELL platform, using the XEON-optimal configuration for this loop leads to a reduction in performance equal to only 75% of the sequential execution performance, a rather poor result.

As can be seen from this simple example, selecting the correct number of threads has significant impact on performance. Furthermore, the optimal number of threads varies from architecture to architecture. In other words, a target-specific thread allocation heuristic tuned for a particular architecture is not portable to other architectures. The next section will introduce a portable predictive modelling technique that can adapt to diverse platforms and can significantly outperform target-specific approaches.

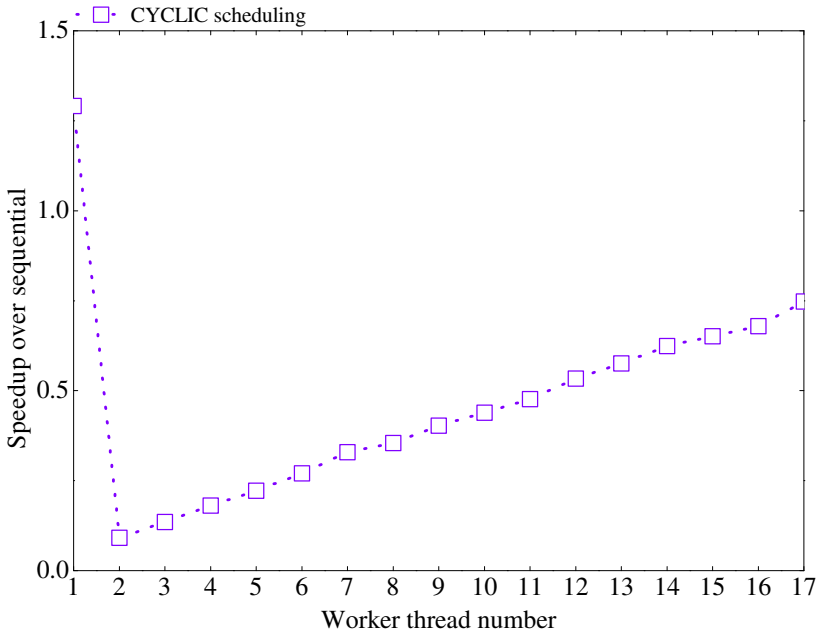


Figure 5.2: Speedup of the parallel loop in the NAS FT benchmark on the CELL processor with the best scheduling policy (i.e. CYCLIC) over the sequential execution. The best strategy is to run one work thread on the PPU with the master thread. This figure shows that it is important to choose the right number of threads.

5.3 Predictive Modelling

Rather than relying on fixed heuristics, this chapter proposes using machine learning as a means to construct automatic and portable models to determine the numbers of threads and scheduling policies for data parallel loops. This section introduces how to build and use the models.

5.3.1 Characterising the Problem

The key point of our machine-learning-based (ML-based) approach is to build a predictor that predicts scalability of a loop and the best scheduling policy for a particular number of threads. The predicted scalability and scheduling policies can be used to determine a parallel configuration for the loop with any input.

The ML-based approach develops two predictors: a data sensitive (DS) predictor and a data insensitive (DI) predictor. The DS predictor is proposed to handle parallel loops whose behaviours are sensitive to the program input while the DI predictor is proposed to tackle those that are not. The only difference between the two predictors is the necessary number of runs needed for profiling (as discussed in section 5.3.3).

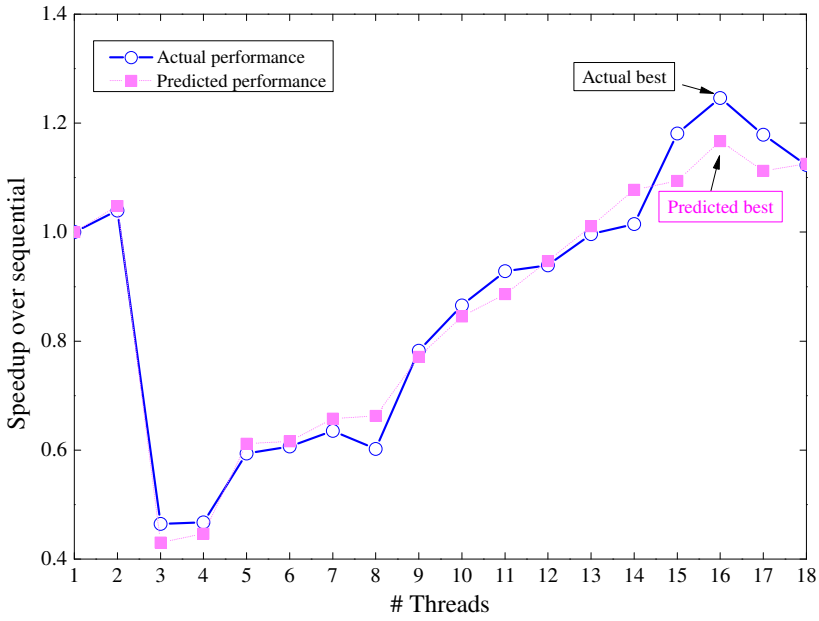


Figure 5.3: Using the artificial neural network model to predict the best number of threads. This example shows the actual and predicted performance of one loop in the NAS *SP* benchmark on the CELL platform. The neural network predicts the speedup of the loop with different number of threads and then picks the number of threads that gives the best predicted speedup.

The problem can be broken down into two sub-problems. The first is to determine the scalability of a data parallel loop, so that we can decide the best number of threads by checking how the loop scales with different numbers of threads. The second is to predict the best scheduling policy for the loop with a given number of threads. The first problem is essentially a regression problem and the second one is a classification problem (see chapter 2). Therefore, we use two different machine learning techniques to solve the problems.

5.3.2 Model Construction

We use a feed-forward artificial neural network (ANN) [Bishop, 2006] to solve the scalability fitting problem and a support vector machine (SVM) (that is similar to the one proposed in chapter 4) to predict the scheduling policy. There are alternative approaches to build a prediction model, such as analytic [Blagojevic et al., 2008] and regression-based [Barnes et al., 2008] schemes, and we compare our ML-based approach against these schemes in section 5.5. Since chapter 4 has provided detailed discussions of a similar SVM-based approach, this section mainly focuses on describing the ANN model.

Table 5.1: Program Features

Static Features	Dynamic Features
IR Instruction Count	Data Access Count
IR Load/Store Count	Instruction Count
IR Branch Count	Branch Count
Loop Iteration Count	L1 Data Cache Missed Rate
	Branch Missed Rate
	Parallel Execution Time

As illustrated in figure 5.3, the ANN model predicts the scalability of a program and then selects the number of threads that gives the best predicted performance. We use the neural network because it has been proven successful on modelling both linear and non-linear regression problems and it is robust to noise [Bishop, 2006]. Our neural network has three layers: an input layer, a hidden layer and one output layer. The activation functions (see chapter 2) of the hidden layer and the output layer are a sigmoid and a linear functions, respectively. The model is trained using the Bayesian back-propagation algorithm [Bishop, 2006] with a number of iterations up to 1,000.

5.3.3 Program Features

The ML-based approach uses code, data and runtime features as means to build an accurate predictor. This section describes how the essential program features are extracted.

Table 5.1 describes the program features used by the two predictors. Since some program features have been proven to be useful for the work presented in chapter 4, we simply reuse them in this work. In addition to these features, we also use dynamic data features to capture input-dependent program behaviours.

Our feature extractor uses source to source instrumentation to collect features. It has a relatively low overhead (less than 25%) compared to a dynamic binary profiling approach which can lead to slowdowns of the original program by factors as much as 10 to 100 times [Qin et al., 2008]. Other approaches, such as dynamic instrumentation approaches [Luk et al., 2005] or profiling-driven auto-parallelisers (as we do in chapter 4), could also be used without affecting the prediction accuracy of our approach.

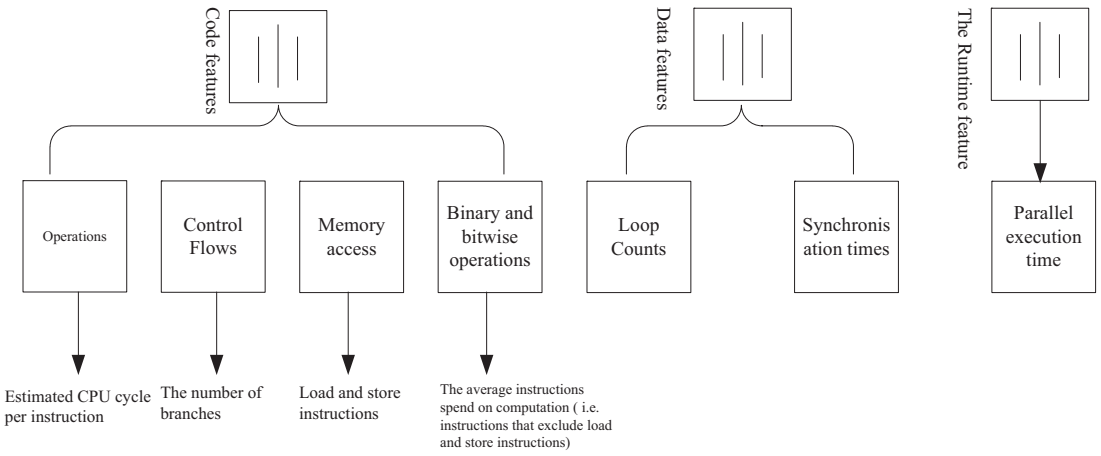


Figure 5.4: Program features are summarised into groups. For example, we statically estimate the number of cycles for all operations and then calculate the average cycle per instruction. We also calculate the number of instructions spent on computation (i.e. instructions that exclude memory load and store operations) and then average it over all instructions of the loop body.

Code Features

In our setting, static code features are derived directly from the source code. Instead of using this high level information directly, our feature extractor post-processes and groups them into four separated groups to capture the essential characteristics of a program, as shown in figure 5.4.

Data Features

Dynamic data features are obtained through profiling runs. The feature extractor instruments the program and then records the number of loop iterations for all parallel loops by executing the instrumented code. During the profiling run, it also records the L1 data cache and the branch miss rates using hardware performance counters.

The `DI` predictor only needs to run the instrumented code once with the *smallest* available input. Using the profiling information, it predicts the dynamic features for a *new* input. The feature prediction is based on a simple linear function:

$$f_{new} = f_{smallest} \times \frac{iterations_{new}}{iterations_{smallest}} \tag{5.1}$$

where f_{new} is the set of new features to be predicted, $f_{smallest}$ is the set of features of the smallest data set which is obtained through the profiling run, $iterations_{new}$ is the number of iterations of the loop when running with the new input, and $iterations_{smallest}$ is the number of iterations of the loop when running with the smallest available input.

The basic assumption of the DI predictor is that at a fine-grain level (i.e. loop level), the program behaviour is relatively stable and predictable across data sets. Of course, there may be occasions where the behaviour of a program is heavily dependent on the input data set. Hence, we also propose the DS predictor that can handle this situation. In contrast with the DI predictor, the DS predictor uses one profiling run of the program for each input to accurately capture dynamic data features. This translates to profiling with only 7% and 1% of all possible parallel configurations on the XEON platform and the CELL platform, respectively. Obviously, the DS predictor is likely to have better performance than the DI predictor because it uses more accurate data set features.

Runtime Features

Our ML-based predictors have to understand how the backend compiler optimises and compiles the target program as "black-box" systems. We simply collect this information by executing the program with one parallel configuration. The profiling run can be done by executing the compiled parallel program with an *arbitrary* number of processors and any scheduling policy, that allows the user to find a trade-off between hardware resources and the time spent on profiling. In this chapter, we execute the program with the CYCLIC scheduling policy and in order to demonstrate that the performance of our approach is not dependent on a particular number of threads, we use two different settings to execute the program on the two multi-core platforms. On the XEON platform, we run the program by using the maximum number of processors, because this setting has the lowest profiling overhead in most cases. On the CELL platform, we run the program by using one SPU, because the SPU must be used by a single thread exclusively and minimising the usage of SPUs is desired.

Summary of Features

Our ML-based approach uses code, data and runtime features to characterise a given program. Code features are directly extracted from the program source; data features are collected through one or more profiling runs of the instrumented program; runtime features are obtained by using the backend compiler to compile the parallel program, executing the compiled program once and recording its execution time.

A surprising result is that the DI predictor produces fairly accurate results by only using one profiling run with the *smallest* program input. As outlined earlier, this is due to the fact that, at a loop level the program behaviours are fairly stable across

Predictor	Profiling with the Sequential Program	Profiling with the Parallel Program
The regression-based model	N	$M \times N$
MMGP	N	$M \times N$
Data sensitive predictor	N	1
Data insensitive predictor	1	1

Figure 5.5: Number of profiling runs needed by each model with N input data sets and M scheduling policies. The ML-based predictors need the least number of profiling runs.

different program inputs. There are some cases where the program behaviours are sensitive to inputs, therefore, we also propose the DS predictor to handle these situations. Figure 5.5 shows the number of profiling runs needed by each predictor. As can be seen from the table in this figure, for a new program, the DI predictor only needs one profiling run regardless of how many input data sets are to be considered.

5.3.4 Training

We use supervised learning algorithms to build our two predictors. Training is performed *off-line* whereby both the ANN and the SVM models are presented with pairs of program features and desired outputs. The desired outputs are generated through repeated, timed execution of training programs with all possible parallel configurations, and then recording the best configurations. This allows the models to learn how to predict the scalability and the scheduling policy from empirical evidence. The collected data are processed into input features for the ANN and the SVM models. Each model finds a class of functions that closely maps the input feature vectors to their corresponding outputs in training data. More precisely, the ANN model finds a predictive function that takes the program features as its input and predicts the program scalability; the SVM model tries to separate a number of classes—where each class represents a particular scheduling policy that gives the best performance for the loop when it runs with a specific input. Since the process of data generation and model training is performed *off-line*, it can be done "at the factory" before the compiler is shipped to the end users.

5.3.5 Deployment

Once the predictors are in place, they can be used to make predictions for *new*, *unseen* programs. For an input program, the predictors firstly collect its program features at compilation time. Then, for each data parallel loop of the program, they predict its

scalability and the best scheduling policy for a given input at runtime. Finally, the predictions are used to determine the parallel configurations for a data parallel loop across different inputs.

5.4 Experimental Methodology

This section introduces platforms, compilers, benchmarks and the evaluation methodology, which are used in the experiments.

5.4.1 Experimental Platforms

In order to demonstrate its portability, the ML-based approach was evaluated on two different multi-core platforms. The first platform is a shared memory homogeneous machine with two quad-core Intel XEON processors, supporting up to 8 threads. The second, in contrast, is a QS20 CELL blade, a disjoint memory heterogeneous system that is integrated with two CELL processors, supporting up to 17 worker threads—of which one worker thread runs on the PPU and the remaining 16 on each of the SPUs. In the experiments, we used the same hardware and software configurations as we used to evaluate the previous ML-based model in chapter 4. In brief, we used the Intel ICC compiler on the Xeon platform and the IBM XLC compiler on the CELL platform to compile the benchmarks.

5.4.2 Benchmarks

The performance of two predictors is evaluated by using 20 programs from three benchmark suites, which are Mibench [Guthaus et al., 2001], NAS parallel benchmark (NPB) [Bailey et al., 1991] and UTDSP [Lee., 1992]. Table 5.2 lists the benchmarks used in our experiments. These parallel workloads represent widely used computation kernels from embedded, high performance, and commercial domains. Some applications from the UTDSP and Mibench benchmark suites were omitted in the experiments because there are loop-carried dependence on their primary time-consuming loops. Moreover, most programs in Mibench have function-pointers and pointer indirection that cannot be properly handled by the IBM XLC compiler; for this reason, we also skipped these programs. This is due to the multiple address spaces of the IBM CELL processor, which prevents us from carrying out experiments on the whole Mibench benchmark suite. As for programs from different benchmark suites that have similar

Table 5.2: Programs used in the experiments.

Suite.Program	Suite.Program
Mibench.stringsearch	Mibench.susan_c
Mibench.susan_e	NPB.BT
NPB.CG	NPB.EP
NPB.FT	NPB.IS
NPB.LU	NPB.MG
UTDSP.Csqueeze	UTDSP. compress
UTDSP.edge_detect	UTDSP.fir
UTDSP.histogram	UTDSP.iir
UTDSP.latnrm	UTDSP.lmsfir
UTDSP.lpc	UTDSP.mult

semantics, only one of them was kept in the experiments. For instance, the *fft* benchmark from Mibench was not included because the *FT* program from NPB is also a fast Fourier transform application. By only keeping programs that have distinct semantics, we have made sure that the predictors were always making predictions for an *unseen* program. Finally, some benchmarks contain multiple loops that have the same code structure and therefore only one representative loop was chosen for experimenting.

In the experiments, we have selected both the primary and trivial parallel loops from the benchmarks. We applied the DI and the DS predictors to predict the parallel configurations for the selected loops on the two platforms. Using individual loops instead of the whole program in the experiments allows us to compare our approach to the upper bound performance. Otherwise, it would not be possible for us to find the upper bound performance at a whole-program scale due to the combinatorial blow-up of parallel configurations.

5.4.3 Comparisons

On the CELL platform, we compared the ML-based predictors against two recently proposed thread allocation schemes: an analytical-based model—Model of Multi-Grain Parallelism (MMGP) [Blagojevic et al., 2008] and a regression-based model [Barnes et al., 2008]. Since MMGP only targets the CELL processor, we were not able to evaluate it on the XEON platform. This section provides a brief introduction of the two approaches.

MMGP

Blagojevic et al. formulate the MMGP model to predict the execution time of a parallel application on the CELL platform [Blagojevic et al., 2008]. This model is defined as:

$$T = a \cdot T_{HPU} + \frac{T_{APU}}{p} + C_{APU} + p \cdot (O_L + T_S + O_C + p \cdot g) \tag{5.2}$$

The parameters in equation 5.2 are described as follows. a is a parameter that accounts for thread and resource contention in the PPU. T_{HPU} is the time spent on the PPU for offloading computation to SPUs. T_{APU} is the task execution time running with one SPU and C_{APU} is the execution time of non-parallelised parts of a task. p stands for the number of SPUs used. O_{Ol}, T_{CSW} and O_{COL} are the overhead of send-receive communication, thread context switching and global synchronisation, respectively. Finally, g is the latency of the workload distribution.

In MMGP, some parameters are program-independent and in the experiments we used the values proposed by the authors. For other program-dependent parameters, we use profiling runs of the program to obtain them.

Regression

Barnes et al. propose a regression-based approach to predict the execution time of a data parallel program [Barnes et al., 2008]. This approach predicts the execution time, T, of a program on p processors by using several profiling runs of the target program with p_0 processors—a small subset of processors, where p_0 can be $p/2, p/4$ or $p/8$. This model aims to find a set of coefficients, $(\beta_0, \cdots, \beta_n)$, of n observations, $(x_1, \cdots, x_n)$, of the input program running with n input data sets on q processors. This is done by using a linear regression for a log form of the execution time—$log_2(T)$ which is formulated as:

$$log_2(T) = \beta_0 + \beta_1 log_2(x_1) + \ldots + \beta_n log_2(x_n) + g(p) + error \tag{5.3}$$

where $g(p)$ can be either a linear function or a quadratic function when the application is running with p processors.

Once the coefficients, $(\beta_0, \cdots, \beta_n)$, have been determined, we can use equation 5.3 to predict the execution time of the program with p processors and select a number of processors that gives the best predicted performance as the ideal number of threads for the input program.

As highlighted by the authors, using a larger number of processors to collect profiling information does not always give a better result than using a smaller number

of processors. For example, when making predictions for p processors, executing the program with $p/2$ processors may give a worse prediction than using $p/4$ processors. Therefore, we use the profiling configuration, $p_0 = p/4$, given by the author in the experiments. For each program, we used two forms of the $g(p)$ function to make predictions and then selected the best performing one as the prediction result of this model.

Predicting Scheduling Policies

This chapter considers four OPENMP scheduling policies: STATIC, CYCLIC, DYNAMIC and GUIDED, which are explained in chapter 2.

Although the MMGP approach and the regression-based model do not predict the best scheduling policy directly, they can be easily extended to fulfill this demand. However, as shown in figure 5.5, they need more profiling runs than the ML-based predictors. To select the ideal scheduling policy, these two models must predict the execution time of a parallel program with different scheduling policies. This means that for each new program, both approaches have to execute the program with each scheduling policy and record the execution time. So that they can predict how the program scale with different numbers of threads when a particular scheduling policy is applied. By ranking the predicted performance, they can choose the best parallel configuration for a given program.

To provide a fair comparison, in the experiments we assumed that these two approaches always choose the right scheduling policy, despite the fact that this is hard to be achieved in reality.

5.4.4 The Evaluation Methodology

The ML-based model was evaluated using "leave-one-out-cross-validation" as described in chapter 2.

For the purpose of evaluating the performance of the ML-based model over different input data, we have randomly generated 60 inputs for most programs and 5 to 10 inputs for some programs because of their inherent constraints—for instance, the *FT* benchmark in NPB requires the input sizes to be powers of two. In this chapter, we present the prediction accuracy of each program as a performance gap to the upper

Table 5.3: Maximum speedups for the largest data set.

Loop	XEON	CELL	Loop	XEON	CELL
Mibench.stringsearch	5.31	1.00	Mibench.susan_d.L1	6.38	3.99
Mibench.susan_d.L2	6.02	1.57	Mibench.susan_e.L1	2.90	1.66
Mibench.susan_e.L2	6.02	1.00	Mibench.susan_e.L3	7.39	1.11
NPB.BT.L1	2.10	1.19	NPB.BT.L2	1.94	2.41
NPB.BT.L3	4.56	5.85	NPB.CG.L1	1.00	1.96
NPB.CG.L2	7.33	1.00	NPB.CG.L3	1.00	1.00
NPB.EP	7.99	6.50	NPB.FT.L1	1.00	1.96
NPB.FT.L2	2.92	1.00	NPB.FT.L3	6.30	7.26
NPB.IS	1.38	1.63	NPB.LU.L1	2.82	7.34
NPB.LU.L2	6.11	1.00	NPB.MG.L1	1.84	1.00
NPB.MG.L2	2.38	1.00	NPB.MG.L3	1.00	1.18
NPB.SP.L1	2.26	1.20	NPB.SP.L2	1.00	5.16
NPB.SP.L3	3.84	7.79	UTDSP.compress	1.00	1.00
UTDSP.edge_detect	7.49	3.00	UTDSP.fir	5.99	4.83
UTDSP.histogram	1.91	1.94	UTDSP.iir	1.00	1.00
UTDSP.latnrm	2.20	1.00	UTDSP.lmsfir	1.42	1.00
UTDSP.lpc	7.03	1.00	UTDSP.mult	7.80	1.50
UTDSP.Csqueeze	7.39	1.86			

bound performance, ε, which is defined as:

$$\varepsilon = (1 - \frac{1}{n}\sum_{i=1}^{n}\frac{T_i^{up}}{T_i^A}) \times 100\% \tag{5.4}$$

where T_i^{up} and T_i^A are the upper bound performance and the performance delivered by approach A for the data set i, respectively. This is a "lower is better" metric and zero means approach A delivers the best performance. For each program, we report the average performance across different inputs.

Upper Bound In order to obtain the upper bound performance, we have exhaustively compiled and executed each program with *all* possible parallel configurations and then selected the best performing one as the actual optimum. Therefore, the gap between the prediction and the actual optimum was calculated based on the *actual* performance of each benchmark.

5.5 Experimental Results

This section first evaluates the maximum performance achievable from selecting the best number of threads and the scheduling policy and, as such, provides an upper-bound on performance with which to evaluate the ML-based approach. It then evaluates

the ML-based predictors against the OPENMP runtime default scheme across data sets showing that it consistently outperforms it. Next, it compares the ML-based approach against two recently proposed models on two platforms. Finally, it evaluates the profile overhead required by each of these approaches and shows that our predictors deliver the best performance and reduce profiling costs by a factor of at least 4.

5.5.1 Upper Bound Performance

Table 5.3 shows the upper bound speedup with the largest data set for each program relative to the sequential version on both the XEON and the CELL platforms. In some instances it is not profitable to parallelise so the respective upper bound speedups are 1. In other cases speedups up to 7.99 and 7.79 are achievable on the Intel and the CELL platforms respectively.

As we found in the experiments, there is no single scheduling policy that consistently outperforms across architectures. On the Xeon platform, the STATIC scheduling policy performs pretty well on most of the benchmarks. The RUNTIME scheduling policy in general is the worst policy on the Xeon platform, which is 2.7 (up to 42) times slower than the STATIC policy. On the CELL platform, however, the RUNTIME scheduling policy can significantly outperform the STATIC policy on some programs. For about 20% of the benchmarks, the RUNTIME policy is on average 1.8x faster than the STATIC policy because of the improvement of load-balance between heterogeneous cores. Moreover, for 15% of the benchmarks, CYCLIC is the best scheduling policy which leads to up to 2.8x speedups over the STATIC policy. This is due to the improvement of cache and TLB behaviours.

Considering the number of threads with scheduling policies together, on average, the worst parallel configuration results in 6.6 (up to 95) and 18.7 (up to 103) times performance degradation on the XEON and CELL platforms respectively. So there is significant performance improvement available, but it is important that we make the right decision.

5.5.2 Comparison with the Default Scheme

This section compares the ML-based predictors with the OPENMP runtime default scheme on the two different platforms.

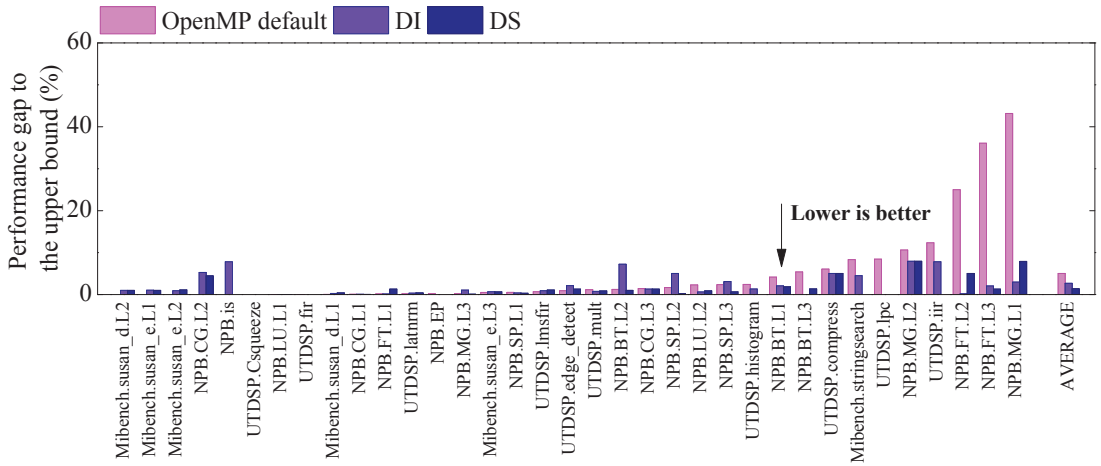


Figure 5.6: Performance gaps to the upper bound on the XEON platform. Performance of the ML-based predictors is more stable and better than the OPENMP runtime default scheme across programs.

Performance Comparison

Figures 5.6 and 5.7 show that ML-based predictors not only have better mean performance but also have better stability across programs compared to the OPENMP runtime scheme. This is manifested by its closed performance gap relative to the upper bound (4%), which translates to 96% of the upper bound performance on both platforms. The ML-based approach outperforms OPENMP's default scheme which achieves 95% and 75% of the upper bound performance on the XEON and the CELL platforms, respectively. These figures show that the ML-based approach adapts to different programs, data sets and platforms.

XEON Platform On the XEON platform (as shown in figure 5.6), on average, the ML-based predictors achieve modest performance improvement when compared with the Intel ICC's OPENMP runtime default scheme. This is mainly because the Intel ICC compiler has been well tuned for the parallel loops that are used in our experiments on the XEON platform [Xinmin et al., 2003]. The ICC default scheme still results in large performance degradation for some programs such as NPB.*FT.L2*, NPB.*FT.L3* and NPB.*MG.L1*, in which its performance gaps to the upper bound are within a range between 43% to 75%. By contrast, the ML-based approach delivers stable performance across data sets. This is exhibited by the performance gap of the ML-based scheme relative to the upper bound performance, which is less than 5% for any benchmark in the experiments; in other words, it achieves at least 95% of the upper bound performance

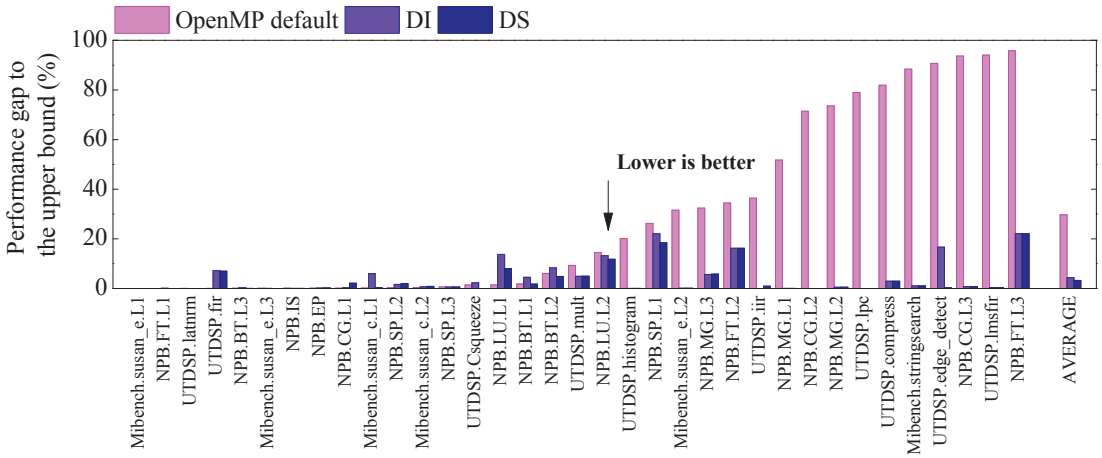
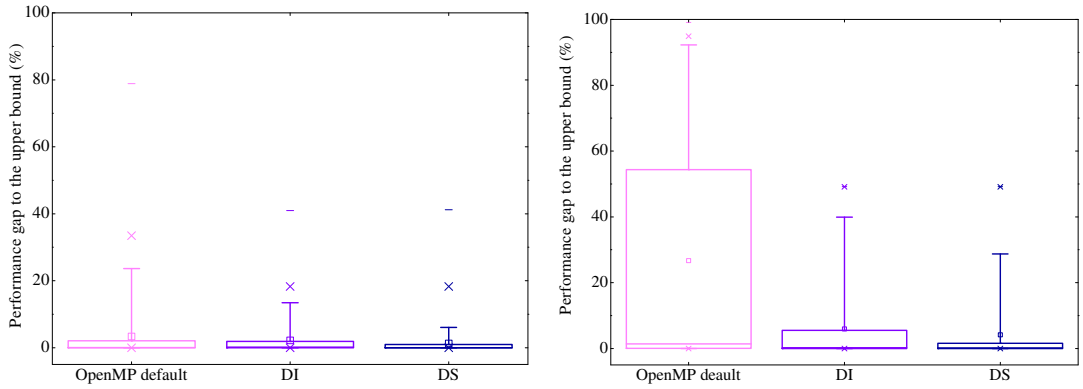


Figure 5.7: Performance gaps to the upper bound on the CELL platform. Again, performance of the ML-based predictors is more stable and better than the OPENMP runtime default scheme across programs.

for all benchmarks. Compared to the OPENMP runtime default scheme, our ML-based approach achieves more stable application performance across programs.

CELL Platform Figure 5.7 compares the performance of the OPENMP runtime scheme against the ML-based approach. Unlike the XEON platform, the OPENMP default scheme does not deliver high performance on the CELL platform. By contrast, the ML-based approach delivers similar performance levels as it does on the XEON platform. On average, the ML-based approach achieves better performance than the OPENMP runtime scheme. The performance gaps of the DI and the DS predictors to the upper bound are only 4% and 3%, respectively. In other words, the DI and the DS predictors deliver 96% and 97% of the upper bound performance, respectively. Compared to OPENMP's default scheme that only achieves 70% of the upper bound performance, these are great improvements. In one case, NPB.*CG.L3*, the performance improvement is 17.5 times greater than the XLC OPENMP runtime default scheme. For some programs, the XLC default scheme has slightly better performance than our approach, but the performance improvement is very small (2.6% on average). Again, the performance of the ML-based predictors is stable across programs on this platform. They deliver above 80% of the upper bound performance for most programs. The OPENMP runtime default scheme, however, has performance below 80% of the upper bound performance on 16 out of 35 programs, in which it achieves 70% of the upper bound performance and in the worst case, it achieves only 4% of the upper bound performance. It is disappointing that the OPENMP runtime default scheme uses all the



(a) XEON (b) CELL

Figure 5.8: Performance gaps to the upper bound per data set.

processors while resulting in poor performance in almost half of the programs. That is to say, the OPENMP default scheme is not stable across programs. As can be seen from our experiments, the best parallel configurations are quite complex on the CELL processor due to the heterogeneous and distributed memory characteristics of the processor and as a result, a fixed heuristic cannot guarantee robust performance across programs and data sets.

Stability Comparison

Box-plots in figure 5.8 summarise the performance gaps to the upper bound for the OPENMP default scheme and the `ML`-based predictors across programs and data sets. The longer a ”whisker” in the diagram is, the less the performance of a model is. The graph clearly shows that performance of the `ML`-based predictors is not only stable across programs and data sets, but also stable across architectures. It also shows that performance of the `DS` predictor is more stable than the `DI` predictor. This is because the `DS` predictor uses additional profiling runs to obtain accurate dynamic data features, which leads to better performance across multiple data sets.

Though the `ML`-based predictors make no assumption of the underlying architecture, they learn the essential program and architecture characteristics from the training data automatically. This is the strength of a predictive model—it frees developers from spending tremendous effort on manually re-tuning a mapping heuristic for a new platform.

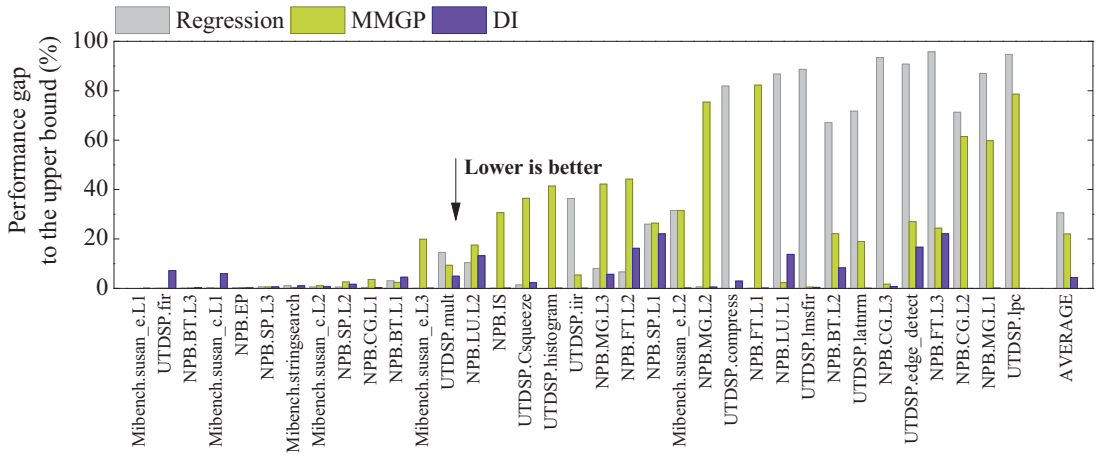


Figure 5.9: Performance gaps to the upper bound on the XEON platform on the CELL platform. The DI predictor has the closest performance to the upper bound.

5.5.3 Comparison with other Models

This section compares the ML-based approach against the regression-based model and the MMGP scheme. This experiment assumes that the MMGP approach and the regression-based model always choose the best scheduling policy, while the DI and the DS predictors must predict what the best policy is.

CELL Platform On the CELL platform, the ML-based approach was compared with MMGP—a performance model that is specialised for the CELL processor, and the regression-based model. Consider the graph in figure 5.9, where the DI predictor[1] has a better average performance as well as greater stability across the programs compared to both MMGP and the regression-based model.

In this experiment we assumed that the MMGP approach and the regression-based model choose the actual optimal scheduling policy for a given number of threads, which means that ideally they should produce better performance results than our predictive models. On the contrary, on average, our ML-based predictors outperform these two models. This is confirmed by the narrow performance gaps to the upper bound performance achieved by our ML-based predictors, which translate to 96% (for the DI predictor) and 97% (for the DS predictor) of the upper bound performance; whereas the MMGP approach and the regression-based model only achieve 76% and 69% of the upper bound performance, respectively. Furthermore, the ML-based predictors achieve stable performance for most programs except for two loops: one is from *FT* and the

[1]The DS predictor has better performance than the DI predictor but is not shown to aid clarity.

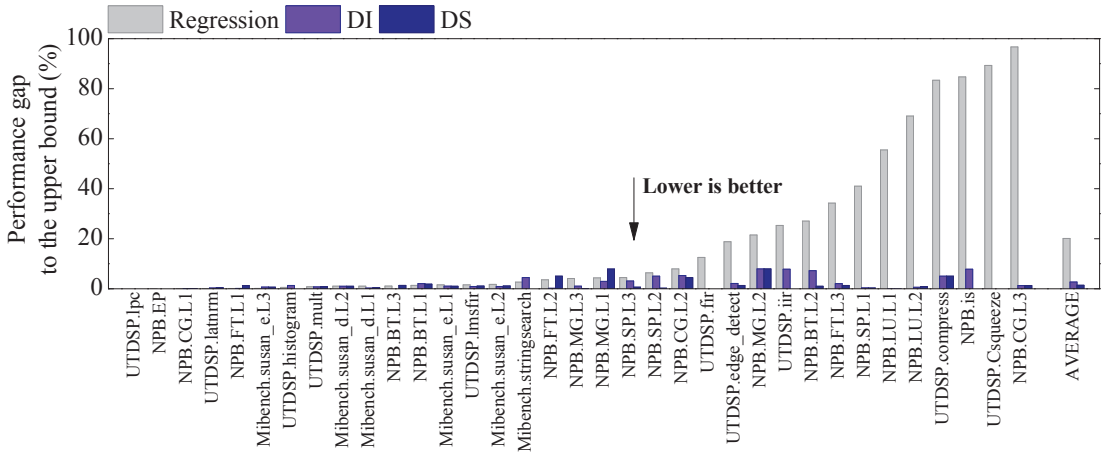


Figure 5.10: Performance gaps to the upper bound on the XEON platform. The machine learning based predictors have the closest performance to the upper bound.

other one is from *LU*. The relatively low performance of these two loops is caused by their unique characteristics that are not covered by the training examples. This problem can be easily solved by using more training programs for the training phase. In contrast to the ML-based predictors, the regression-based model shows the poorest prediction result that achieves just 4% of the upper bound performance, which translates to a wide performance gap of 96% to the upper bound; and the MMGP scheme's poorest prediction achieves only 18% of the upper bound, which translates to a performance gap of 82% to the upper bound.

XEON Platform We only compared the DI and the DS predictors with the regression-based model on the XEON platform, because MMGP does not target this platform. According to figure 5.10, the ML-based predictors have consistently better performance and greater stability across programs compared to the regression-based model. On average, the ML-based predictors outperform the regression-based model which only delivers 80% of the upper bound performance. Although we have assumed that the regression-based model always picks the best scheduling policy, it still slows down some programs significantly. In one example, NPB.*CG.L3*, it delivers only 3.3% of the upper bound performance. The poor performance is caused by failing to choose the correct number of threads.

Table 5.4: Profiling overhead of each model.

Model	Intel	CELL
DI predictor	4.79 secs	4.80 mins
DS predictor	13.10 mins	1.75 hours
Regression	59 mins	16.45 hours
MMGP	NA	41 hours

5.5.4 Summary of Performance Results

Sections 5.5.2 and 5.5.3 demonstrate two advantages of the ML-based approach when compared with the OPENMP runtime default scheme and the analytical-based approaches (i.e. the MMGP approach and the regression-based model). Firstly, the ML-based approach has the best performance on average. Secondly, its performance is stable across programs, data sets and architectures.

5.5.5 Profiling Cost

Profiling cost is a critical measurement when evaluating any predictive model. This section discusses profiling overhead of our ML-based predictors, the regression-based model and the MMGP approach.

The DS predictor requires additional profiling runs and as a result, it executes the program with 7.18% and 1.16% of all possible configurations on the XEON and the CELL platforms, respectively.

Table 5.4 shows that the ML-based predictors have the smallest profiling costs on both platforms. According to the table, the profiling overhead of the DI predictor is very low and can be neglected. Though the DS predictor requires additional profiling runs, when compared to the regression-based model, it still reduces the profiling overhead by factors of 4 and 12 times on the XEON and the CELL platforms, respectively. Comparing to the MMGP scheme, the DI predictor and the DS predictor significantly reduce the profiling overhead by factors of 29 and 512 times, respectively.

Figures 5.11 and 5.12 show the profiling cost per program for each model (the profiling cost of the DI predictor is very low and is not shown in this figure to aid clarity). The profiling cost of the DS predictor is within ranges from 0.2% to 13% and from 0.08% to 4.7% of all possible configurations on the XEON and the CELL platforms, respectively. Overall, the ML-based predictors have lower profiling cost than

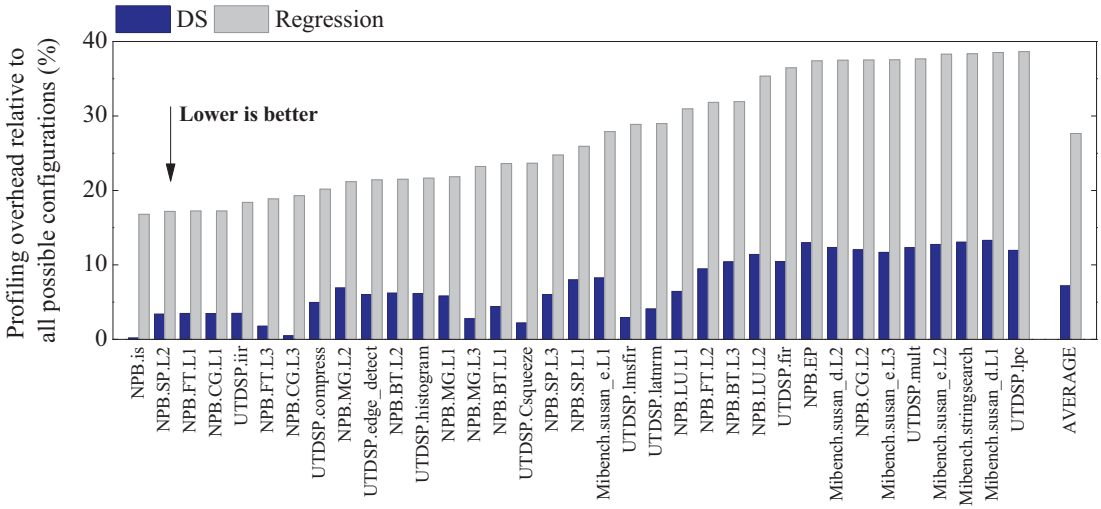


Figure 5.11: Profiling overhead for each model on the XEON platform. The ML-based predictors have the lowest profiling overhead.

the MMGP approach and the regression-based model, which have to use many more profiling runs to make a prediction.

Summary of Profiling Overhead

As can be seen from table 5.5, the ML-based approach requires a fixed number of profiling runs regardless of how many threads and scheduling policies are considered. This is an important feature that allows our approach to scale with the advances of future multi-core processors and runtime systems.

The regression-based model has fairly expensive profiling overhead because it does not reuse prior knowledge. Thus, every time it makes predictions for a new program, it starts with no prior knowledge and has to obtain and learn behaviours of the new program through profiling runs. The MMGP approach suffers from the same problem of having to use expensive profiling runs to characterise the behaviour of the input program. The reason the MMGP approach produces more accurate predictions than the regression-based model is because its formula consists of architecture-dependent features. However, to obtain these features requires detailed human-knowledge about the underlying hardware. In contrast to these schemes, the ML-based approach automatically learns from the training data. By utilising prior knowledge learned from the training data, the ML-based predictors require the least profiling overhead for a new program and achieve the best performance.

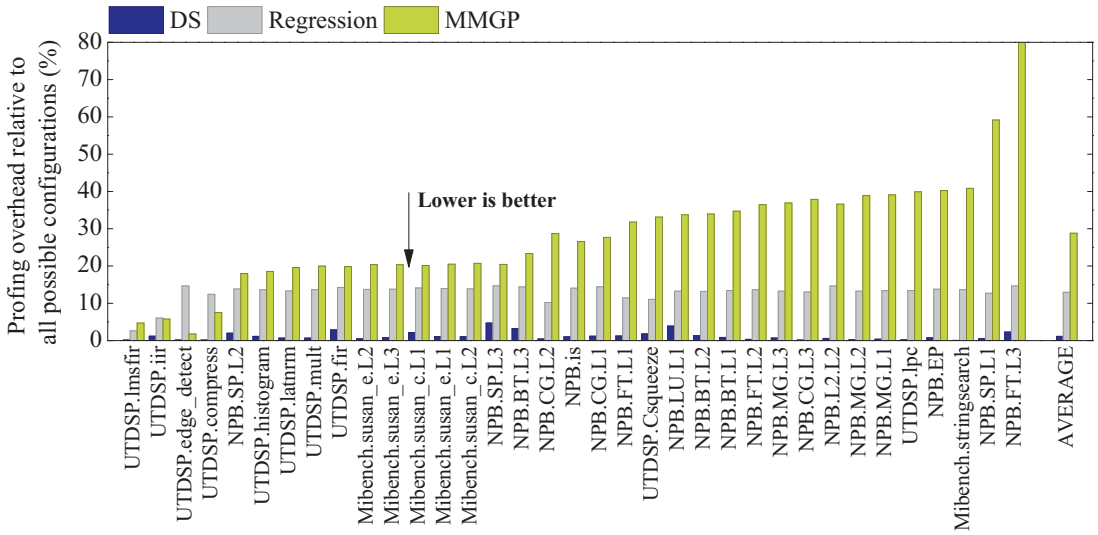


Figure 5.12: Profiling overhead on the CELL platform. The ML-based predictors have the lowest profiling overhead.

5.6 Discussion

To characterise the input, the DS predictor has to profile the sequential program with each input data set once. This is a method that has been widely used in iterative compilation techniques [Kisuki et al., 1999]. Compared to traditional iterative compilation techniques, the DS predictor has the advantage of avoiding search the whole compilation space. Though the profiling overhead of DS can be amortised by the future, repeated runs of the input program, it may be considered expensive. The overhead can be reduced by using sampling techniques [Sherwood et al., 2002], e.g. only profiling a few loop iterations.

On the other hand, as can be seen from section 5.5, the DI predictor achieves similar performance results while having much cheaper profiling overhead compared to the DS predictor. From this point of view, the DI predictor is a stronger technique compared to the DS predictor in terms of applicability.

5.7 Conclusions

This chapter has presented two portable machine-learning-based (ML-based) predictors to predict the best numbers of threads and scheduling policies of a data parallel loop across multiple program inputs.

The ML-based predictors were evaluated by comparing them against two state-of-the-art performance prediction models and the OPENMP runtime default scheme. Experimental results on two different multi-core platforms (Intel XEON and IBM CELL) show that the ML-based predictors not only produce the best performance on average but also have the most robust performance across programs, data sets and architectures. Compared to two recent models, the ML-based predictors significantly reduce the profiling cost for a new program. The ML-based approach is different from prior approaches in that it allows the compiler designer to automatically construct an optimising compiler which can generate various parallel configurations for data parallel loops across different input data sets and platforms.

The techniques presented in this chapter and chapter 4 provide a comprehensive compiler-based solution to map data parallelism onto multi-core processors. The next chapter will address the problem of mapping streaming parallelism—a more complex parallel paradigm that contains task, data and pipeline parallelism.

Chapter 6

Mapping Streaming Parallelism

Chapters 4 and 5 have proposed the use of machine learning to map data parallelism onto multi-cores. This chapter now considers the mapping problem of streaming parallelism, a common parallel pattern that contains task, data and pipeline parallelism.

This chapter is organised as follows. Section 6.1 first introduces the work conducted in this chapter. Then section 6.2 motivates the need for an automatic and adaptive approach for partitioning streaming parallelism. This is followed by a description of the machine-learning-based approach in sections 6.3 and 6.4. Experimental methodology and results are discussed in sections 6.5 and 6.6, respectively. Finally, section 6.7 summarises the main points of this chapter.

6.1 Introduction

Streaming parallelism is a common parallel pattern existing in many application domains such as audio, video, network and signal processing. These application domains are rich in parallelism and the program dependencies and parallelism can be explicitly identified with high level streaming languages such as StreamIt [Gordon et al., 2002], Stream C/KernelC [Kapasi et al., 2003] and Brook [Buck et al., 2004]. Programs written with a streaming language are called stream programs. A stream program can be viewed as a composition of parallel tasks, which is represented as a stream graph. Each node in a stream graph represents a computation task and arcs between nodes represent communication between tasks. When compiling a stream program, the compiler must perform whole-program optimisations on the stream graph so as to generate an efficient mapping [Gordon, 2010].

Typically, the compiling and executing of stream applications is broken into two stages: partitioning the stream graph into regions which are allocated to threads and then scheduling, which allocates the threads to the underlying hardware. This is by no means a new challenge and there is an extensive body of work on mapping task, data and pipeline parallelism to parallel architectures using runtime scheduling [Ramamritham and Stankovic, 1984], static partitioning [Sarkar, 1991; Subhlok et al., 1993; Kwok and Ahmad, 1999], analytic models [David et al., 1996; Navarro et al., 2009], heuristic-based mappings [Gordon et al., 2006], or integer linear programming (ILP) solvers [Kudlur and Mahlke, 2008; Udupa et al., 2009]. They can each achieve good performance but are restricted in their applicability. Fundamentally, such approaches are based on the developers' view about the most significant costs of the target platform and typical programs, encoding a hardwired human-derived heuristic. However, as shown in this chapter later, the best form of the program varies across programs and platforms.

To overcome the problem of hardwired heuristics, many researchers have proposed the use of machine learning (ML) as a methodology to automatically construct optimisation strategies [Moss et al., 1998a; Stephenson et al., 2003]. Such an approach has the advantage of being portable across different platforms without requiring expert knowledge. However, until now it has been limited to relatively straightforward problems where the target optimization to predict is fixed, e.g. determining compiler flag settings [Hoste and Eeckhout, 2008], loop unroll factors [Stephenson and Amarasinghe, 2005] or the number of threads per parallel loop (as presented in chapter 5). Determining the best partitioning of a stream program is fundamentally a more difficult task. First of all, rather than predicting a fixed set of optimisations we are faced with predicting an unbounded set of coalesce and split operations on a program graph. As the graph changes structure after each operation, this further increases the complexity. A secondary issue limiting the applicability of machine learning is its reliance on sufficient training data. This problem is particularly acute for emerging parallel programming languages where the application code base is small as is the case for streaming languages.

This chapter tackles both problems by developing a ML-based model to automatically derive a good partition for StreamIt [Thies et al., 2002] programs on multicores, making no assumption on the underlying architecture. Rather than predicting the best partitioned graph, it develops a *nearest-neighbour* based machine learning model that predicts the ideal partitioning *structure* of the StreamIt program. It then

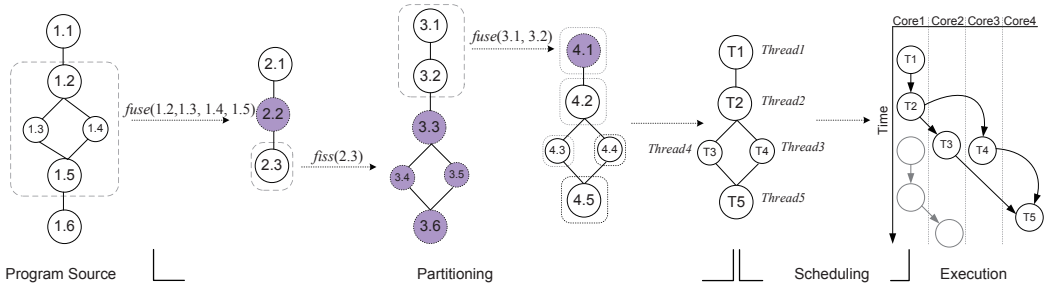


Figure 6.1: The mapping and executing process can be broken into two main stages partitioning and scheduling. Partitioning is responsible for mapping nodes to threads. This is achieved by a series of fuse and fission operations on the original source program. At the end of this process, each node of the final graph is allocated to a thread and so the original graph has been partitioned. Scheduling allocates each of these threads to cores. Scheduling may be dynamic especially if the number of threads is greater than cores. This chapter focuses solely on partitioning.

searches through a program transformation space (without executing any code) to find a program of the suitable structure. This chapter also presents a micro-kernel stream program generator to overcome the problem of insufficient training programs. The generator is able to provide many small training examples for the machine learning model.

6.2 Background and Motivation

This section first defines the problem of stream graph partitioning. It then shows that finding a good partition for a streaming application is highly dependent on the program and the underlying hardware platform.

6.2.1 Background

Our `ML`-based partitioner is evaluated with StreamIt benchmarks. In StreamIt, each program is represented by a stream graph. It is the compiler's responsibility to partitioning this graph and allocate partitions to threads which are then scheduled on the underlying hardware. This is a two-stage process and is illustrated in figure 6.1. First, the nodes in the original graph are merged into larger nodes or spilt into smaller nodes by a sequence of fuse and fission operations. This gives a transformed program where each node is allocated to a separate thread. Each thread is then scheduled to the hardware by the runtime. The first stage is called *partitioning* as it is concerned with

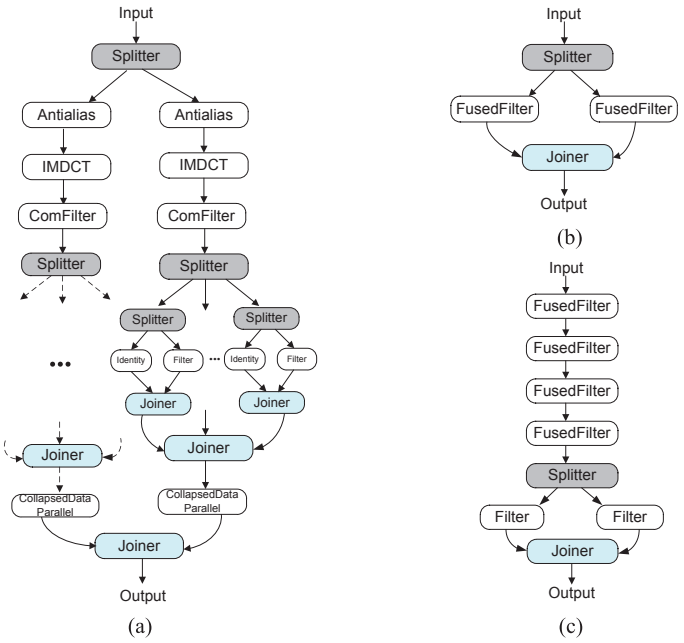


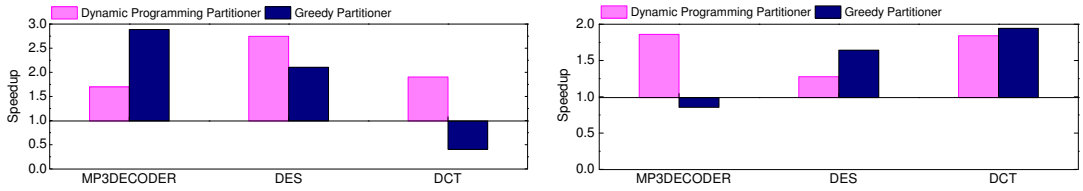
Figure 6.2: A simplified stream graph of the *MP3DECODER* STREAMIT program (a). Each node is a task that can be executed in pipeline fashion. Concurrent task parallelism is achieved after each splitter node. A greedy partitioner applied to this program gives the graph (b) with just four nodes. A dynamic programming based partitioner gives the graph (c) with 8 nodes.

determining those regions of the stream program that will be eventually allocated to a thread. The second stage is called *scheduling* and is responsible for the allocation of threads to processors. In this chapter, we are interested in the mapping of nodes to threads and hence focus purely on the first stage of the process[1], partitioning.

6.2.2 Motivation

Finding a good partitioning for a streaming program is difficult due to the large number of possible partitions. Figure 6.2 (b) and (c) show two possible partitioned versions of the original *MP3DECODER* program shown in figure 6.2(a). Both are obtained by applying a sequence of fuse and fission operations on the original graph. The first partitioning shown in figure 6.2(b) corresponds to a greedy partitioner, the second partitioning, figure 6.2(c) corresponds to a dynamic programming based method; both are STREAMIT compiler built in heuristics [Thies et al., 2002]. The question is which is

[1]Note: as the proposed approach uses a machine learning approach, it implicitly considers the behaviour of the scheduler along with the rest of the underlying system (hardware, operating systems etc.) when generating training data.



(a) 4-core platform (b) 8-core platform

Figure 6.3: The relative performance of two partitioning schemes with respect to a naïve partitioning scheme. The results are shown for two platforms and three distinct STREAMIT programs. Greedy partitioning performs well on *MP3DECODER* on the 4-core platform but not as well as dynamic programming partitioning on the 8-core. This relative ordering is reversed for *DES* and *DCT*. Determining the best partitioning depends on program and platform.

the best one? This problem of graph partitioning in its general form is known to be NP-complete and it is difficult to devise a general heuristic [Bui and Jones, 1992].

To illustrate this point, consider figure 6.3, which shows the performance of each partitioning approach on two different multi-core platforms: a 2x dual-core (4-core) machine and a 2x quad-core (8-core) machine. On the 4-core the greedy scheme performs well for *MP3DECODER*; it has a lower communication cost, exploiting data parallelism rather than the pipeline parallelism favoured by the dynamic programming partitioner. On the 8-core platform, however, the dynamic programming-based heuristic delivers better performance as load balancing becomes critical.

When examining two further programs, *DES* and *DCT*, it is clear that the best partitioning algorithm for a particular machine is reversed. The figure shows that there is no current ”one-fits-all” heuristic and the best heuristic varies across programs and architectures. Rather than relying on heuristics, we would like a scheme that automatically predicts the right sequence of fuse and fission operations for each program and architecture. In the case of *MP3DECODER*, this means a good strategy will select the operations that give the partitioned code in figure 6.2(b) for 4-cores and the partitioned code in figure 6.2(c) for 8-cores. However, predicting the correct sequences of fuse and fission is highly non-trivial given the unbounded structure of the input program graphs.

The next section describes a novel approach to partitioning streaming applications. Rather than predicting the sequence of fuse and fission operations directly, it tries to predict the right *structure* of the final partitioned program. Given this target structure, it then searches for a sequence of fuse and fission operations that generates a partitioned

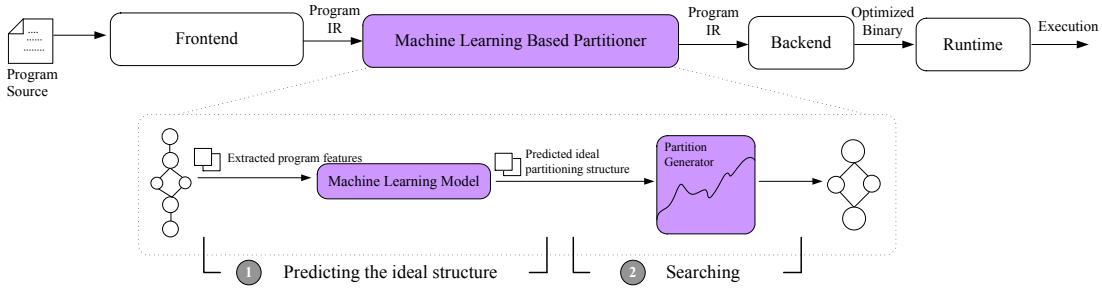


Figure 6.4: Work flow of the machine learning based compiler framework. The compiler takes in program source code and produces an optimised binary. In the middle of the compiler is a ML-based partitioner. The partitioner firstly predicts the features of an ideal partitioning structure of the input program. This is done by checking the similarity of the input program features to prior knowledge. Then, it searches the transformation space to generate a program whose features are as close as possible to the predicted ideal structure.

program that fits the predicted structure as closely as possible.

6.3 Predicting and Generating a Good Partition

One of the hurdles in predicting the best sequence of fusion and fission operations is that the graph keeps changing structure after each operation. In figure 6.1, the second operation fiss(2.3) would have to be renamed fiss(1.6) if the first operation (fuse(1.2,1.3,1.4,1.5)) had not taken place. Any scheme that tries to predict a sequence of fuse and fission operations has therefore to take into consideration the structure of the graph at each intermediate stage. The supervised predictive modelling schemes explored to date are incapable of managing this [Duda et al., 2000]. This chapter takes a different approach. Instead of trying to predict the sequence of fuse and fission operations, the problem is divided into two stages as illustrated in figure 6.4:

1. Predict the ideal structure of the final partitioned program.

2. Search a space of operation sequences that delivers a program as close as possible to the ideal structure.

As illustrated in figure 6.5, our 2-step approach actually performs on two different spaces. On the *program space*, an ideal predictor would take in the input program graph and directly predict the output program graph. However, it is impossible to

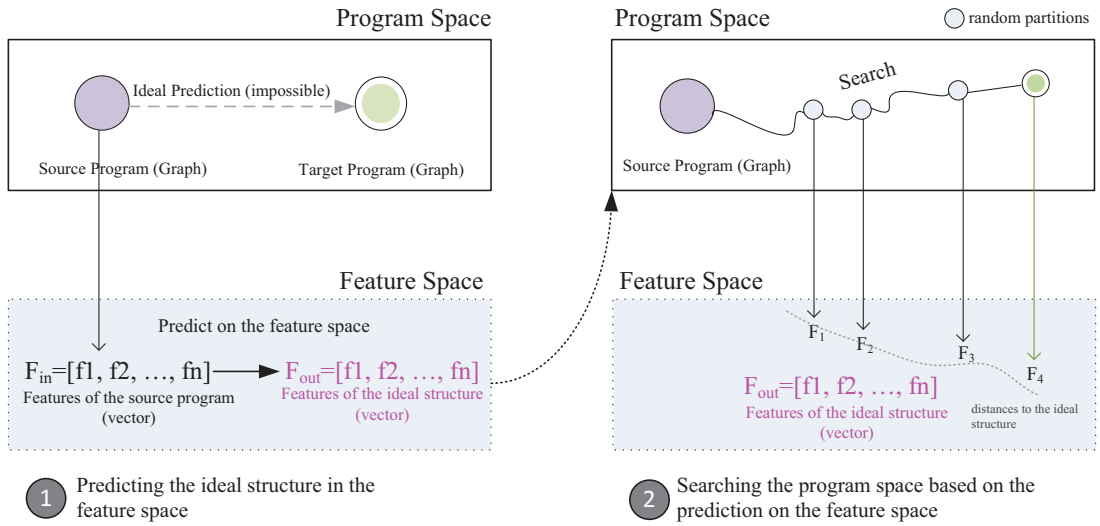


Figure 6.5: The ML-based approach performs on two different spaces. It first predicts the feature vector of the ideal partitioning structure on the *feature space*. Then it searches the *program space* to generate a partition which is as close as possible to the ideal structure. The distance to the ideal structure of a generated partition is measured on the feature space.

predict unbound graph structures using supervised learning. Instead our ML-based approach first translates the prediction problem onto the *feature space* and then maps the predicted result back to the program space using random search. In the first step, the ML-based model predicts the ideal partitioning structure on the feature space. It takes the feature vector of the input program and predicts the feature vector of the ideal program structure. On the feature space the prediction is performed on fixed feature vectors, which enables us to build a supervised learning model. Unfortunately there are no functions that can map the predicted feature vector back to a legal partition on the program space. To generate an output program, in the second step our approach first performs random search on the program space to generate many legal partitions. It then measures the distance to the ideal structure for each generated partition on the feature space and selects one partition whose distance is as close as possible to the ideal structure.

In essence, the first stage focuses on determining the goal of partitioning, i.e. the structure of the partitioned program without regard to how it may be actually realised. The second stage explores different legal operation sequences until the generated partitioned program matches the goal. This frees us from the concern of correctly predicting the syntactically correct sequence of fuse and fission operations. Instead the compiler can try arbitrary sequences until it reaches a partition that closely matches the goal.

The next section describes how to predict a good partitioning goal and is followed by a section describing how the machine learning based compiler can generate a sequence of fuse and fission operations to reach that goal.

6.3.1 Predicting the Ideal Partitioning Structure - Setting the Goal

Ideally, the proposed scheme needs to predict the ideal partitioned structure of any input graph program. In order to cast this as a machine learning problem, it is required to build a function f which, given the essential characteristics or features X_{orig} of the original program predicts the features of the ideal partitioned program X_{ideal}. Building and using such a model follows the well-known three-step process for supervised machine learning [Bishop, 2006]: (a) generate training data (b) train a predictive model (c) use the predictor. Training data is generated by evaluating (executing) randomly generated partitions for each training program and recording their execution time. The features of the original and best partitioned program are then used to train a model which is then used to predict the best ideal partitioning structure for any *new*, *unseen* program. One of the key aspects in building a successful predictor is developing the right program features in order to characterise the original and goal programs. This is described in the next section and is followed by sections describing training data generation, building the predictor using nearest neighbours and then using the predictor.

6.3.2 Extracting Features

Rather than trying to deal with unbounded program graphs as inputs and outputs to the predictor, the essential characteristics of such graphs are described by a fixed feature vector of numerical values. The intention is that programs with similar feature vectors have similar behaviour. This assumption is empirically evaluated in section 6.6.3. The predictor uses program *features*, to characterise a streaming application. The set of program features are summarised in table 6.1. Two sets of those features are extracted from the overall stream graph and the critical path of the program. Thereby, one set represents the overall characteristics of a streaming program and the other captures characteristics *solely* of the program's critical path. Features are extracted from the stream graph (i.e. program `IR`) without running the program, thus the overhead of extracting features is insignificant.

For a given streaming program, the model firstly extracts a feature vector, $X_{orig} = [x^1_{orig}, x^2_{orig}, \cdots, x^n_{orig}]$ from the original stream graph. X_{orig} is used to characterise

Table 6.1: Program features extracted from a streaming application.

Program Features	
#Filter	#Joiner
Pipeline depth	Splitjoin width
Avg. unit work[1]	Max unit work
Pipeline work	Splitjoin work
Computation[2]	Computation of stateful filters
Branches per instruction	Load/store per instruction
Avg. dynamic communication rate	Max dynamic communication rate
Avg. commun. rate	Computation-commun. ratio
Avg. commun. / (unit work)	Avg. bytes of commun. / (unit work)
Max commun. rate / (unit work)	Work Balance

[1] The amount of estimated instructions spent on per input data item. This includes the instructions that are used to transfer input and output data items.

[2] The amount of estimated instructions spent on doing computation. This *excludes* the instructions that are spent on transferring input and output data.

it. The same feature vector X_{part} is used to characterise any partitioning of a given program. The features are normalised and we use principal component analysis (PCA) [Bishop, 2006] (see chapter 2) to reduce redundancies between features.

6.3.3 Generating Training Data

Once there is a means of describing the original and partitioned programs, we can start generating training data. Training data are generated by evaluating on average 3000 *different* randomly generated partitions for a program and recording the execution time, t. For each program, the model also extracts program feature sets, X_{orig} and X_{part}, of both the program and its partition respectively. Program features and execution time are put together to form an associate training dataset $T = \{(X^i_{orig}, (X^{i,j}_{part}\, t^{i,j})\} i \in 1, \cdots, N$ and $j \in 1, \cdots, R$, for N training programs in which each program has R different partitions.

Synthetic Stream Program Generation

One particular problem encountered in training for new languages is that the training set is small. There simply is not a large enough program base with which to work. To overcome this problem, this work builds a micro-kernel stream program generator to

generate many small training examples, supplementary to existing benchmarks. This allows training of the predictive model on larger data sets. The stream program generator automatically extracts micro-kernels (i.e. working functions and communicating patterns) from any subset of existing STREAMIT programs. It limits the generated programs to a space with parallel parameters (i.e. pop and push rate, pipeline depth, split-join width, and loop iteration counts etc). Then, it generates a large number of small training examples in which the parallel parameters and working functions are varied.

The cost of generating new benchmarks can be neglected because millions of programs can be generated in an order of minutes. Running programs to generate training data from potentially thousands programs, however, is prohibitively expensive. Therefore, it is desired to select only a limited number of representative programs. We propose a clustering technique to select representative benchmarks, which is also applied to choosing the ideal partitioning structure (see section 6.3.4). Informally, this is done by examining the generated programs and selecting only those which are sufficiently distinct from the existing training set in the feature space. Detailed description of this technique is given in appendix A. Although producing training data takes time, it is only a one-off cost incurred by the `ML`-based model. Furthermore, generating and collecting data is a completely automatic process and is performed off-line. Thus, it requires far less effort than constructing a heuristic by hand.

6.3.4 Building a Model

Once sufficient training data is generated, we are in a position to build a predictive model. The proposed model is based on a straightforward *nearest-neighbour classifier* [Duda et al., 2000]. When train the model, for each training program, the features of the original program X_{orig} and those of its best found partition X_{ideal} are recorded. When used on a new unseen program, the model finds the program from the training set whose features most closely match the new program's features. It then simply returns the features, X_{ideal} of the training program as the predicted best ideal partition for the new program.

In the setting of this chapter, each program in fact has a number of partitions that give good performance. To capture this, the best partitions are grouped into regions using *clustering*. We consider the cluster with the best average performance and select a representative candidate as described in the next section.

Selecting the Ideal Partitioning Structure

Each training program has a number of good partitions. The task is to select the most useful one, i.e. the partition that is likely to be good for similar programs. The good partitions are clustered using a *K-Means* clustering algorithm [Bishop, 2006]. The right number of clusters K is determined by using the standard *Bayesian Information Criterion* (`BIC`) score [Pelleg and Moore, 2000; Schwarz, 1978]. Informally, `BIC` is a measurement of the "goodness of fit" of a clustering parameter (i.e. K) to a given data set. The larger the `BIC` score, the higher chance that we find a good clustering number for the data set. Work in this chapter uses the `BIC` formulation given in [Pelleg and Moore, 2000], which is:

$$BIC_j = \hat{l}_j - \frac{p_j}{2} \cdot logR \tag{6.1}$$

where $\hat{l}_j$ is the log-likelihood of the data when K equals to j, R is the number of points in the data (i.e. the number of generated partitions), and p_j is a free parameter to estimate, which is calculated as: $p_j = (K-1) + dK + 1$ for a d-dimension feature vector plus 1 variance estimate [Sherwood et al., 2002]. $\hat{l}_j$ is computed as:

$$\hat{l}_j = \sum_{n=1}^{K} -\frac{R_n}{2} log(2\pi) - \frac{R_n \cdot d}{2} log(\hat{\sigma}^2) - \frac{R_n - K}{2} + R_n log(R_n/R) + logS_n \tag{6.2}$$

where R_n is the number of points in the nth cluster, $\hat{\sigma}^2$ is the average variance of the distance from each point to its cluster centre, and S_n is the normalised average speedup of the nth cluster.

The K-Means clustering algorithm is applied for the generated programs with different cluster numbers K. For each clustering result, its corresponding `BIC` score is firstly calculated. Then, the clustering number, K_{best}, which gives the highest `BIC` score, is chosen. After this point, we know the partition space can be represented by K_{best} clusters.

Once the number of clusters is found, the proposed method selects the cluster that has the largest number of good partitions. We select the 10% of partitions that are close to the cluster centroid of the selected cluster and normalise their feature values. The normalised features are considered as the ideal partition structure.

Figure 6.6 visually depicts the use of this clustering technique. Each point in the figure represents a good partition of the StreamIt benchmark *LATTICE*. To aid clarity, the original multi-dimensional feature space has been eliminated to a two dimensions space. For these programs, there are three distinct clusters of good partitions.

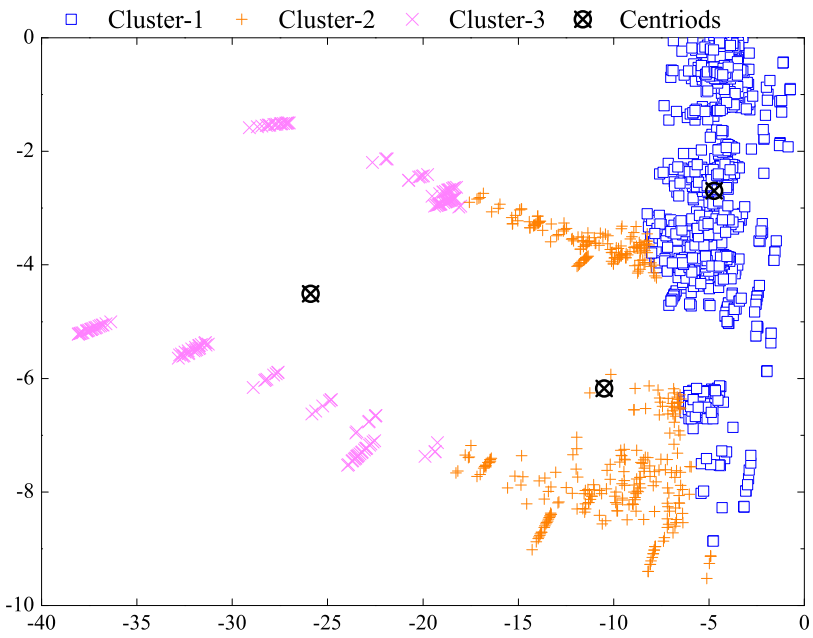


Figure 6.6: The feature space is projected into 2 dimensions in this diagram for presentation purposes. Each of the points represents a good partitioning structure for the *LATTICE* program. Using clustering there are 3 clusters found and centres of clusters are marked as centroids. Cluster-1 is selected as it has the best mean speedup. The features around the centroid of cluster-1 are then averaged and used as the ideal partitioning structure.

Cluster-1 is chosen because it contains 66% of all the good partitions and has the highest mean speedup.

6.3.5 Using the Model to Predict the Ideal Partitioning Structure

Once we have gathered training data and built the model as described above, we can now use the model to predict the ideal partition structure of a *new*, *unseen* program as shown in figure 6.4. The model firstly extracts features of the input program, processes its program features using PCA, and use the *nearest neighbour* model to predict the ideal partitioning structure of the input program. The nearest neighbour scheme picks a program in the training set that is the most similar to the input program. This is done by comparing the input program's features to known programs' features. Once the nearest neighbour has been selected, its ideal partitioning X_{ideal} is used as the predicted ideal structure for the new program. In rare instances, the model may not be able to find any training programs that are similar enough to the input program (i.e. no programs

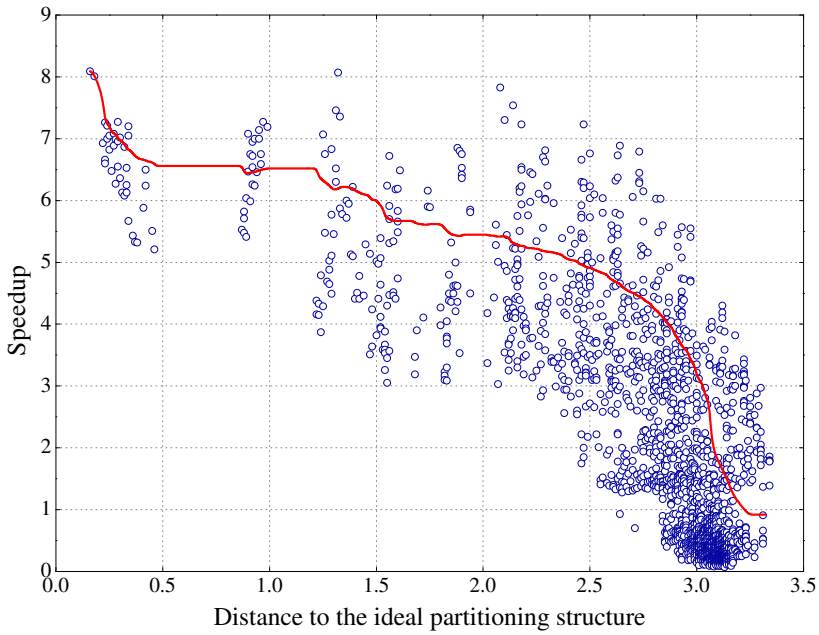


Figure 6.7: Separating partition candidates with Euclidean distances. The x-axis represents the distance to the ideal partitioning structure and the y-axis represents the speedup relative to the STREAMIT default graph partitioner. Each dot represents a partition choice and the line represents the mean speedup of that partition within a distance. The line represents mean speedup of the partitions whose distances to the ideal structure are less than a certain value.

in the training set are close to the new input program). It then simply uses the default partitioner provided by the compiler.

6.4 Searching and Generating a Partition Close to the Predicted Ideal Structure

The previous section provides a means to predict the ideal partition structure without actually running the code. We can now generate new partitions by applying random fuse and fission operations to the input program's graph. For each generated partition, the model measures its Euclidean distance from the predicted ideal structure in the feature space. This is repeated many times, selecting the partition nearest the ideal structure.

Figure 6.7 illustrates the use of distance as a means of determining the best partition candidate for the STREAMIT program *LATTICE*. Each dot represents a unique partition. There are over 3000 *different* partitions of which only 15% of the partitions

are better than the partition generated by the STREAMIT default scheme. Given the large number, selecting a partition to improve is non-trivial. The figure shows that distance to the ideal structure is a good measure of the quality of a partition. If we choose a distance of less than 0.5 as the confidence level, then we will pick a partition that is at least 5.21 times (6.6 times on average) faster than a partition generated by the STREAMIT default partitioning heuristic.

In this work, the `ML`-based model generates on average 3000 potential partitions for each new program, selecting the one that is nearest to the ideal. It does not execute any of these programs, merely extracts their features and measures the Euclidean distance. Each partition takes less than 100 ms to generate and evaluate, so is not a significant cost.

6.5 Experimental Methodology

This section describes the platforms, compilers, and benchmarks used in the experiments as well as the evaluation methodology.

Benchmarks The STREAMIT benchmark suite version 2.1.1 was used to evaluate the proposed approach. These applications represent typical streaming parallel programs containing both pipeline and data parallelism. On average, each program contains 46 (up to 168) filters at the `IR` level.

Compilers The `ML`-based model is implemented as a stream graph partitioner in the STREAMIT compiler (version 2.1.1). The STREAMIT compiler is a source to source compiler which translates the partitioned stream graph into C++ code. The Intel ICC compiler version 11.0 was used to convert the C++ program to binary. The compiler flags used for ICC is "`-O3 -xT -aXT -ipo`".

Hardware Platform The experiments were performed on two multi-core platforms: a 4-core platform (with two dual-core Intel XEON 5160 processors running at 3.0GHz with 8GB memory) and an 8-core platform (with two quad-core Intel XEON 5450 processors running at 3.0GHz and has 16GB memory). The dual-core XEON 5160 processor has a 4MB L2-cache while the quad-core XEON 5450 has a 12MB L2-cache. Both platforms run with 64-bit Scientific Linux with kernel 2.6.17-164 x86_64 SMP.

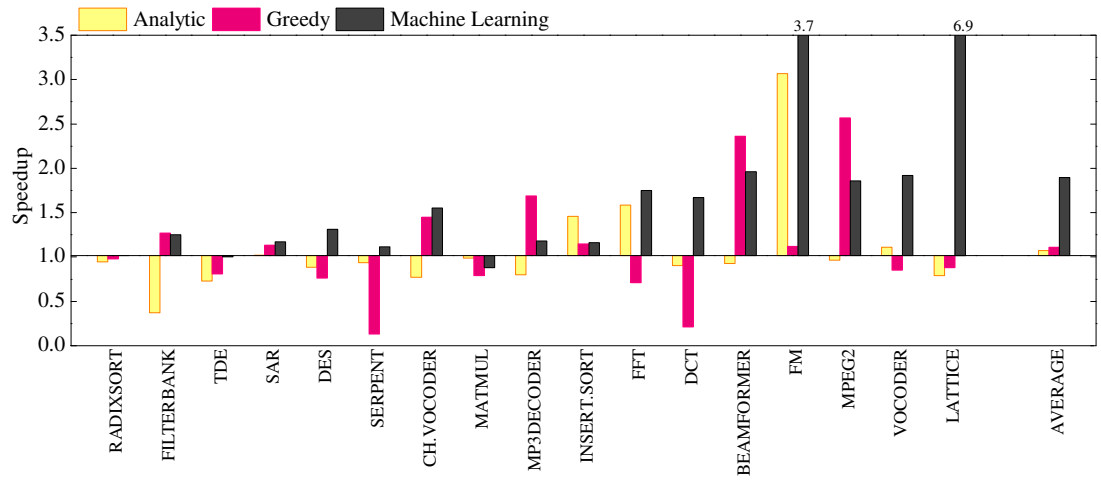


Figure 6.8: Performance comparison on the 4-core platform for the analytical model, the greedy partitioner and the ML-based model.

Cross-Validation Once again, the standard evaluation method–*leave-one-out-cross-validation* was used to evaluate the propose approach [Bishop, 2006]. This means the target program was removed from the training program set and then a model was based on the *remaining* programs. This also guarantees that the benchmark generator has not seen the target program before. The trained model is used to generate partitions for the removed target program. This procedure is repeated for each program in turn.

Synthetic Benchmarks Roughly over 100K synthetic programs were generated by the benchmark generator and around 60 to 300 of them were selected for training. The benchmark generation and selection process takes less than 15 minutes.

6.5.1 Comparison

StreamIt has its default partitioning strategy, a sophisticated dynamic programming based partitioning heuristic [Thies, 2009]. All results are presented relative to this default and it provides a challenging baseline. To provide a wider comparison, we also evaluate a recently proposed analytical-based pipeline parallelism model [Navarro et al., 2009] and an alternative greedy-based heuristic available within the StreamIt compiler [Gordon et al., 2006]. The analytical-based model finds a suitable parallel mapping by predicting the execution time of a given streaming application and it is implemented in the StreamIt compiler. For each program, the analytical-model-based partitioner generates 50,000 partitions of a single program and selects a mapping

which has the best predictive performance as output. The machine learning scheme, in contrast, predicts the best *structure* and selects the partition closest to it, using an order of magnitude fewer candidates.

6.5.2 Best Performance Found

In addition to comparison with existing approaches, we also wish to evaluate the proposed model by assessing how close its performance is to the maximum achievable. However, it is not possible to determine the best, due to the combinatorially large number of partitions. Instead we randomly generated 3,000 *different* partitions for each program and select the best performing partition as an indication of the upper bound on performance that could be achieved if there were sufficient resources. This is called ”Best-Found” performance.

6.6 Experimental Results

This section first reports the performance of the ML-based approach against alternative approaches on the 4-core platform. This is followed by a short explanation of the results generated by different models. Next, the accuracy of the ML-based model is evaluated in predicting good structures and what type of partitioning is important for each program is analysed. Finally, the model is extended to an 8-core platform and evaluated.

6.6.1 Performance Comparisons

Comparison with other Techniques

Figure 6.8 shows the performance results for the 4-core platform. On average the analytical-based and the greedy-based partitioners do not significantly improve over the STREAMIT default dynamic-programming-based partitioner. The ML-based approach, however, is able to deliver significant improvement over the default scheme with a 1.90x average speedup.

Analytic On average, the analytical-based model only achieves 1.07x speedup over the STREAMIT default partitioner. This is not a surprising result because the

STREAMIT default partitioner is a strong baseline tuned by hand. It is only able to improve performance in 4 out of the 17 programs. It successfully partitions *FM* resulting in a 3.0x speedup by coarsening the pipeline. However, it fails to balance the pipeline of *FILTERBANK* due to its inability to capture the complicated communication pattern of the program. This leads to a greater than 2x *slowdown*.

Greedy The greedy partitioner also fails to significantly improve over the STREAMIT default partitioner. On average, it achieves a 10% performance improvement over the default partitioner. In approximately half of the programs, it gives a performance improvement. For example it is able to achieve an impressive 1.69x and 2.57x speedup on *MP3DECODER* and *MPEG2* by reducing communication through aggressive fusion. However, it also slows down 9 applications, particularly in the case of *SERPENT* and *DCT*, which are up to 7.7x slower than the default dynamic programming approach. This result clearly shows the best partitioning heuristic varies from program to program.

Machine Learning The ML-based approach, on the other hand, can greatly improve performance compared to the default partitioner and gives more stable results. It achieves better performance in most of the benchmark, up to 6.9x for *LATTICE*. In just one case, *MATMUL*, it performs worse than the default (as do the other 2 schemes). The backend ICC compiler aggressively auto-vectorises the program which has not been captured by our model. This issue can be solved by adding additional features to the model and is the subject of future work.

Comparison vs. Best-Found Performance

Although the ML-based scheme performs well compared to existing approaches, it is useful to know whether there is further room for improvement. Figure 6.9 shows the comparison of the proposed scheme against the ”Best-Found” performance. For *FM* the ML-based reaches this maximum, but for other programs such as *MP3DECODER*, *INSERTIONSORT* and *VOCODER*, there is significant room for improvement. So although the proposed approach outperformed all prior techniques on *VOCODER*, it could have done better. Overall there is a 2.5x average maximum speedup available and it achieves 60% of that maximum performance.

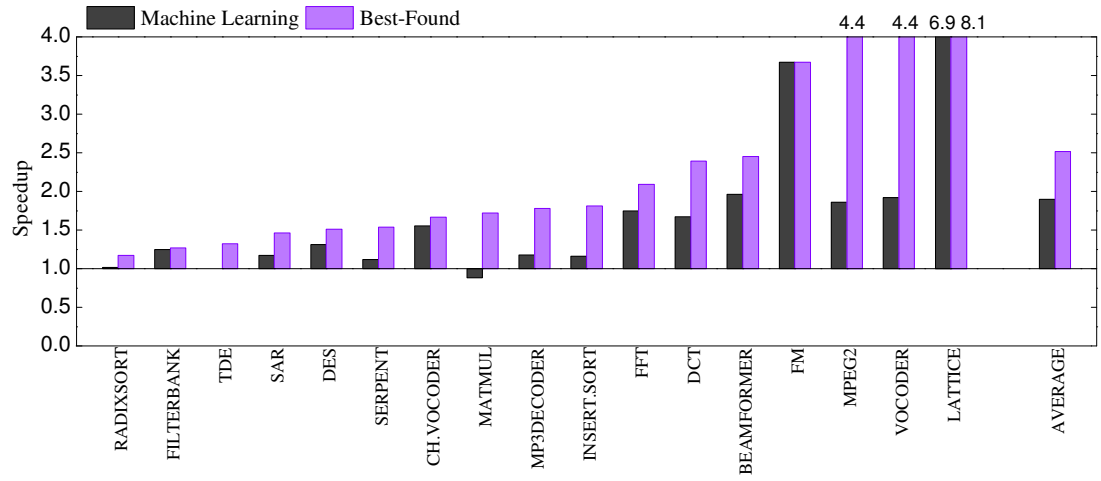


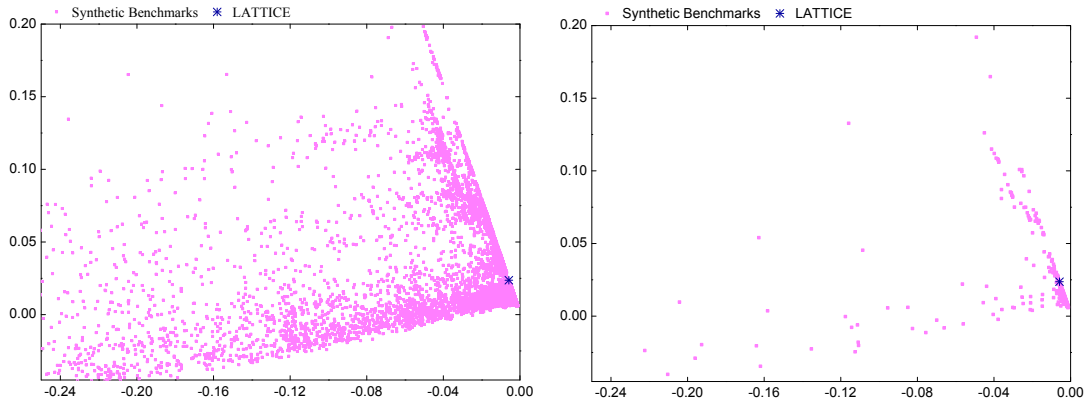
Figure 6.9: The performance of our approach vs the best performance found out of on average 3000 executions per program on the 4-core platform.

6.6.2 Explanation

This section first explains how the benchmark generator works. Then it investigates the partitions generated by different approaches for three selected StreamIt benchmarks and as such, provides further details about the ML-based model and other approaches.

The Benchmark Generator

We explain how the benchmark generator works using a training example. In this case, *LATTICE* is first removed from the training program set, so that *LATTICE* will not be seen by the learning algorithm. In order to build an accurate predictive model, we need to have similar programs in the training program set. To do this, the benchmark generator first extracts micro-kernels from the remaining training programs and then uses the extracted micro-kernels to generate many new, small training examples as shown in figure 6.10(a). By using a statistical method (as described in appendix A) to select representative programs, the generated benchmarks can be represented by a smaller number of programs as shown in figure 6.10(b). As can been seen from this diagram, the representative programs actually cover the unseen program, i.e. *LATTICE*, in the feature space. Using the knowledge obtained from the selected programs, a predictive model is able to generate good partitions for *LATTICE*-similar programs, even though the learning algorithm has not seen exactly the same program before. This example explains why our ML-based model achieves great performance improvement for the *LATTICE* benchmark in the experiments (see figure 6.8).



(a) Synthetic Programs (b) Representative Benchmarks

Figure 6.10: Using the benchmark generator to generate representative training benchmarks. Here, the original multi-dimensional feature space has been projected to a two-dimensional space to aid clarity. In this example, *LATTICE* was first removed from the training program set. Using existing programs, the benchmark generator first produced synthetic programs as shown in (a). The generated programs were then pruned and the selected representative programs are able to cover the unseen program *LATTICE* in the feature space (b).

In figure 6.10(a), some regions where most training programs fall into are denser than others. Consequently, these program-dense regions tend to have more representative benchmarks. This is actually a good feature that enables us to focus on the most popular programs in case the training cannot be performed on an extensive set of training programs.

Generated Partitions

RADIXSORT This application has a regular parallel structure: it is pure pipeline parallelism; 10 out of its 13 filters have exactly the same computation-communication ratio. For this program, both the dynamic-programming based and the greedy-based algorithms give a partition that has the ”Best-Found” performance. Thus, the ML-based approach is not able to improve their results.

LATTICE This application contains 36 filters with both data (i.e. splitjoin) and pipeline parallelism as shown in figure 6.11 (a). Finding a good partition for it is certainly nontrivial and different approaches give different answers. Figures 6.11 (b) to (e) illustrate the partitions given by four partitioners: the STREAMIT default, the greedy-based, the analytical-based, and the ML-based partitioners, respectively. The

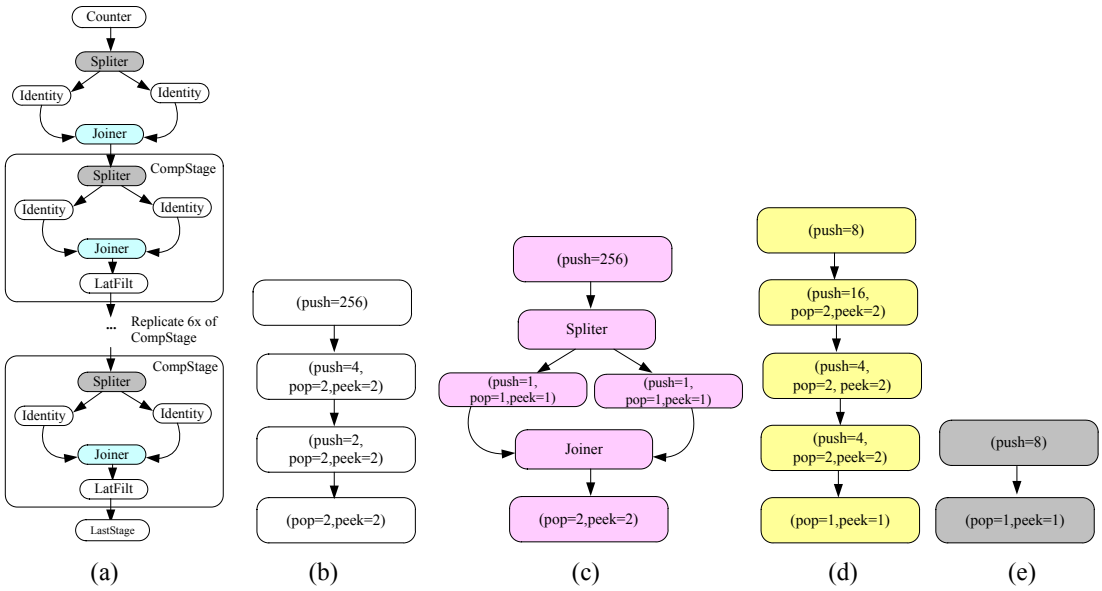


Figure 6.11: The stream graph (a) and partitions generated by different approaches for *LATTICE*. Each box represents a filter and the communication rate of each filter is denoted. The dynamic programming based partitioner gives (b), the greedy partitioner gives (c), the analytical model gives (d), and the ML based approach gives (e) by coarsening the stream graph to reduce communication overhead.

STREAMIT default partitioner aims to form a balanced pipeline and generates a partition with four nodes. This partition outperforms the solutions given by the greedy-based and the analytical model-based partitioners, which generate partitions with more threads at the scheduling stage, bringing extract runtime overhead. In contrast, the ML-based approach generates a coarse-grain stream graph, which has relatively fewer number of threads and lower communication cost. As a result, the ML-based approach achieves better performance than other techniques. By examining this application, it turns out that the computation of *LATTICE* is relatively smaller compared to its communication cost on the 4-core platform. Therefore, a good partitioning strategy will try to coarsen the stream graph to reduce communication overhead. In this case, the ML-based approach identifies the program characteristics of *LATTICE* and applies an appropriate heuristic to aggressively coarsen the stream graph.

VOCODER The stream graph of *VOCODER* has over 120 filters containing large splitjoin sections and long-stage pipelines. Unlike *LATTICE*'s partitioning strategy, merely considering coarsening the stream graph is not the right choice for this application. Figures 6.12 (a) and (b) correspond to the partitions given by the STREAMIT

default scheme and the ML-based approach respectively. This time, the ML-based approach takes a different strategy. In order to reduce communication overhead, it first coarsens those small computation kernels. At the same time, it exploits data parallelism in the critical path (which contributes to around 40% of the total computation) and generates a 9-node partition. When compared with the Best-Found solution, a 13-node partition, the ML-based model could be smarter by predicting a more aggressive partitioning goal.

Summary As indicated by these examples, different partitioning strategies should be applied to applications with different program characteristics (section 6.6.3 gives detailed discussion about the program characteristics). Essentially, the analytical model and the two STREAMIT partitioners are ”one-size-fits-all” strategies. They can be improved by a program-aware partitioning scheme. Developing such a scheme by hand is, however, extremely hard. The ML-based approach, on the other hand, solves this problem by leveraging machine learning techniques. It uses prior knowledge to select and apply the program-specific partitioning strategy according to program characteristics of the target program, resulting in better performance than hardwired heuristics.

6.6.3 Analysis of Results

This section analyses the behaviour of the proposed approach. It first evaluates how accurate the nearest neighbour model is. Next, it evaluates how useful the feature space is in distinguishing good partitions. Then, it examines the structure of the best partitions found and examines what optimisation criteria are important in delivering performance. This is followed by an analysis of the STREAMIT benchmark generator.

Correlation of Program Features

The intuition behind the ML-based model is that similar programs will have similar ideal partitioning structures as long as we are able to have features that capture similarity accurately. Figure 6.13 confirms this assumption. It shows the program features of the ideal partitioned program vs. the features of the original program for each of the benchmark. The original multi-dimensional feature vectors have been projected into a single value for each program to aid clarity. This figure shows a strong correlation between the program features and the ideal partitioning structure. This can be quantified by using the *correlation coefficient* [Bishop, 2006] (see chapter 2 for the definition).

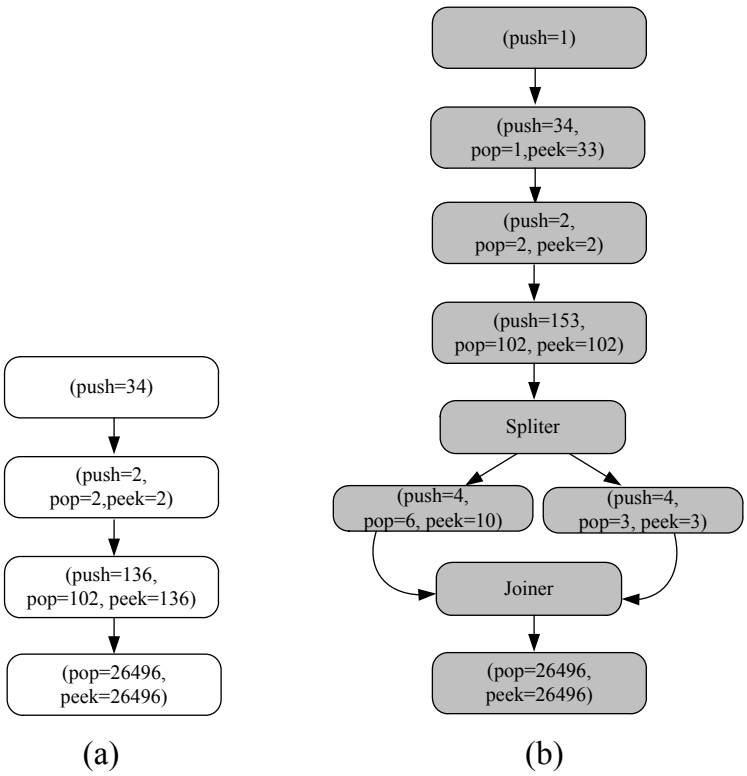


Figure 6.12: Partitions generated by the STREAMIT default method (a) and the ML-based approach (b) for *VOCODER*. Each box represents a filter in which the communication rate is denoted. The STREAMIT default scheme exploits purely pipeline parallelism. The ML-based approach exploits both pipeline and data parallelism.

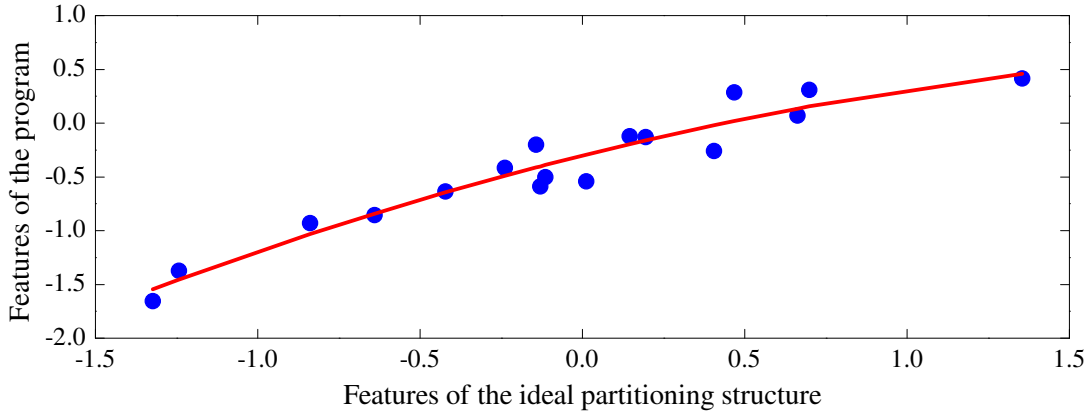


Figure 6.13: Correlation between program features and the ideal partitioning structure for each of the 17 benchmarks. This figure shows there is strong correlation between program features of a program and its ideal partitioning structure. Note that the feature vector of each program has been reduced (by using PCA) into a single value to aid visualisation.

It takes a value between -1 and 1, the closer the coefficient is to $+/-1$, the stronger the correlation between the variables. It is 0.9 in this case, which indicates a high correlation between program features and the ideal partitioning structure. This diagram shows evidence that means the premise for the nearest neighbour model is valid.

Distance-based Mapping Selection

The box plots in figure 6.14 summarise the performance of partitions around a predicted ideal partitioning structure. They show the performance of partitions with a normalised distance of less than 0.5 to the predicted ideal partitioning structure, as used by the predictive model. The diagram shows that regions around the predicted ideal result in good performance. The top and the bottom of the ”whisker” of each program represent the highest and the lowest speedup found in the region around the predicted ideal. For the majority of programs we obtain significant performance improvement if we can generate a mapping that is closer to the predicted ideal partitioning structure. The one exception is *MATMUL*, as seen in figure 6.9. If we zoom in on *VOCODER*, we see that the average speedup obtained in this region is 2.7. The lower value is 1.9 and the upper is 4.4. If we look at figure 6.9, we see that the proposed scheme selects a partition that achieves just a 1.9 speedup–the lower bound, while the best performance is 4.4–the upper bound. This shows that the predictive model could improve if was smarter in choosing the ideal structure within a good cluster for this program.

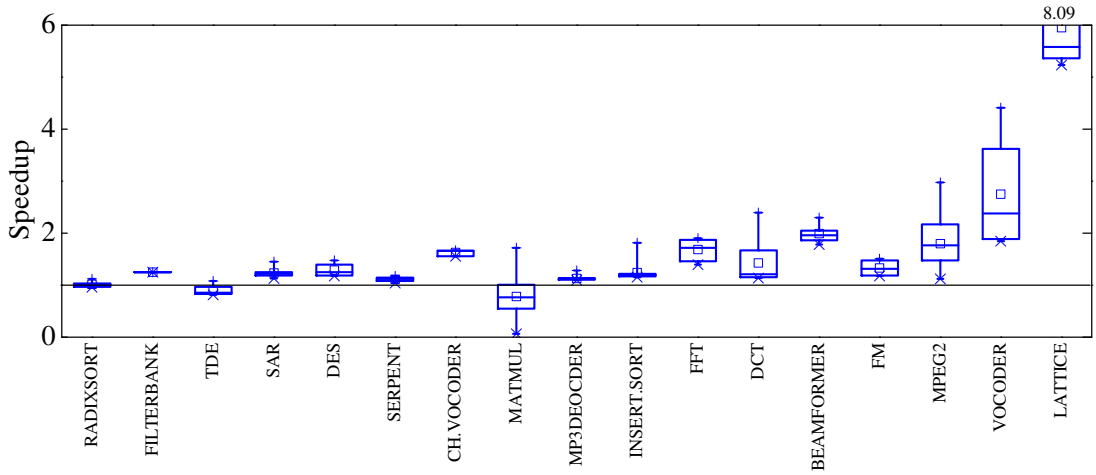


Figure 6.14: Performance of mappings around the predicted ideal partitioning structure. The distance of each program to the predictive ideal partitioning structure is evaluated with weighted Euclidean distance. The x-axis represents the program and the y-axis represents the speedup relative to the STREAMIT default mapping. The central box denotes the mean speedup and the top and bottom of each box represent the highest and the lowest speedup using the ML-based model.

Figure 6.15 shows how the performance of a partitioning structure varies as a function of its distance from the ideal partitioning structure. This diagram averages the results across all benchmarks and shows that partitions near the ideal structure give the best average performance. The figure demonstrates that Euclidean distance from the ideal structure is a useful means of discriminating good partitions from poor.

Importance of Partitioning Choices on Performance

Section 6.2 has shown that the best heuristic varies across programs. We now consider the importance of specific partitioning characteristics for each program on the 4-core platform. We have considered a number of characteristics that a partitioning algorithm may wish to consider in making partitioning decisions e.g. communication computation ratio, average push rate etc. Figure 6.16 shows a Hinton diagram illustrating the importance of a number of different partitioning objectives on the performance of each program. Intuitively, this information gives us an indication of those characteristics on which an optimising heuristic should focus. The larger the box, the more significant the issue for that program is. The x-axis denotes the programs, the y-axis denotes partitioning criteria. Figure 6.16 shows that each of these objectives has an impact on each program. The computation to communication ratio is important for all

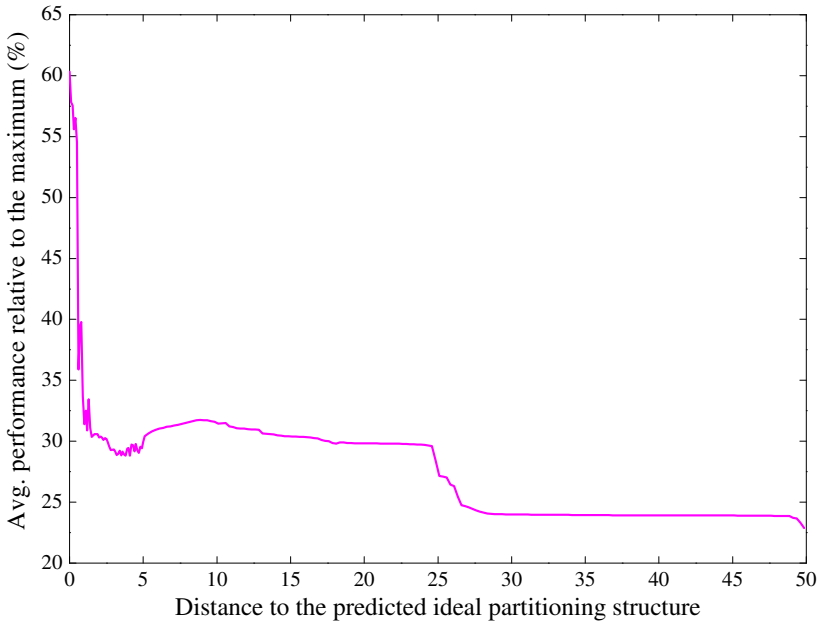


Figure 6.15: Average performance relative to upper bound vs. distance from ideal partition. This figure shows the average performance of a partition as a function of its distance from the predicted ideal structure. As the distance decreases, performance improves.

programs and extremely important for the *CHANNELVOCODER* and *SAR*. Having a balanced pipeline, however, is less important overall. Some programs are sensitive to all of these objectives, e.g. *RADIXSORT* while for some program e.g. *FFT*, one issue, coarsen the splitjoin sections, is of overwhelming importance. This diagram illustrates just how hard it is for a heuristic which typically focuses on one or two objectives, to find the best partitioning for all programs.

6.6.4 Adapting to a New Platform

In order to evaluate the portability of the ML-based model, we have ported it to a 2x Quad-core Intel XEON (8 cores in total) platform. Training data was collected from the new platform and used to train the model. Note that the same program features and methodology were used to train the model as used on the 4-core platform. Due to time constraints, relatively fewer partitions were collected for each training program. This will affect the performance of the model.

Figure 6.17 shows the performance of the different approaches on the 8-core platform. The most striking result is that for some applications, the greedy partitioner does better than it does on the 4-core platform. For *DES* and *FFT*, the greedy partitioner achieves 1.28x and 1.78x speedup respectively compared to the default dynamic pro-

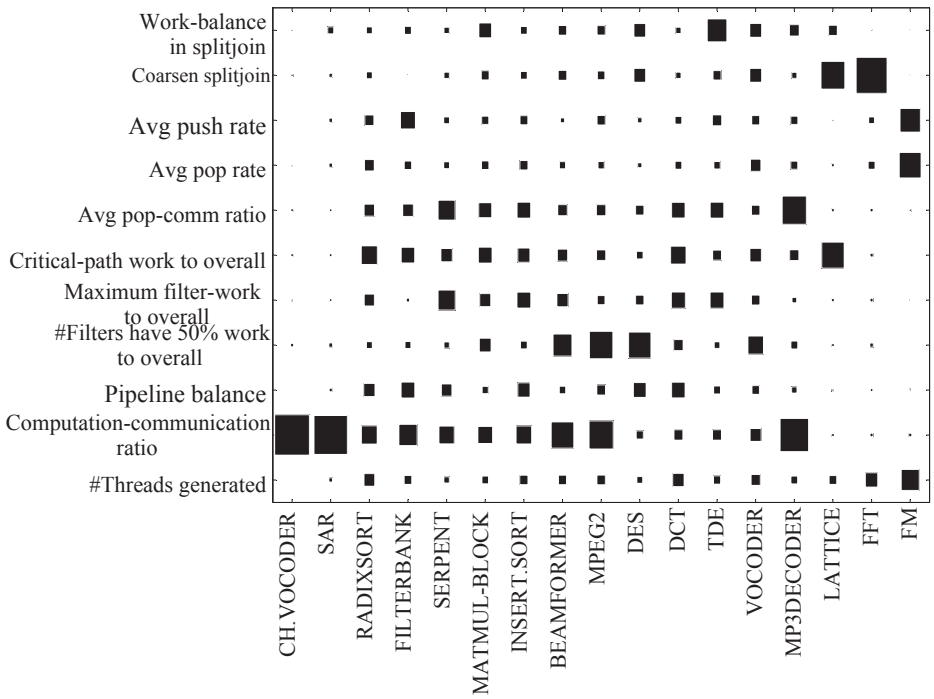


Figure 6.16: A Hinton diagram showing the partitioning objectives that are likely to impact performance of each benchmark. The larger the box, the more likely a partitioning objective affects the performance of the respective program.

gramming partitioner when on the 4-core platform, it slows down these programs to 70% of the partition generated by the STREAMIT default. The greedy partitioner improves the performance of 10 benchmarks on this platform, up to 4.9x for *VOCODER*, with an average 1.2x speedup but gives significant performance slowdowns for 5 programs. The analytical-based model also gives unstable results. It gives an average speedup but only gives noticeable improvement on 4 programs. It, however, gives a significant slowdown on *FILTERBANK* by more than a factor of 2.

In contrast to those two approaches, the ML-based approach is more stable across programs, with just one small slowdown in the case of *RADIXSORT*. On average it achieves 1.8x improvement across the benchmarks. With the correct amount of training data, this is likely to improve even further. Compared to the analytical model and the greedy heuristic, the ML-based model is stable not only across programs but also across platforms. This example demonstrates the portability of the machine learning approach.

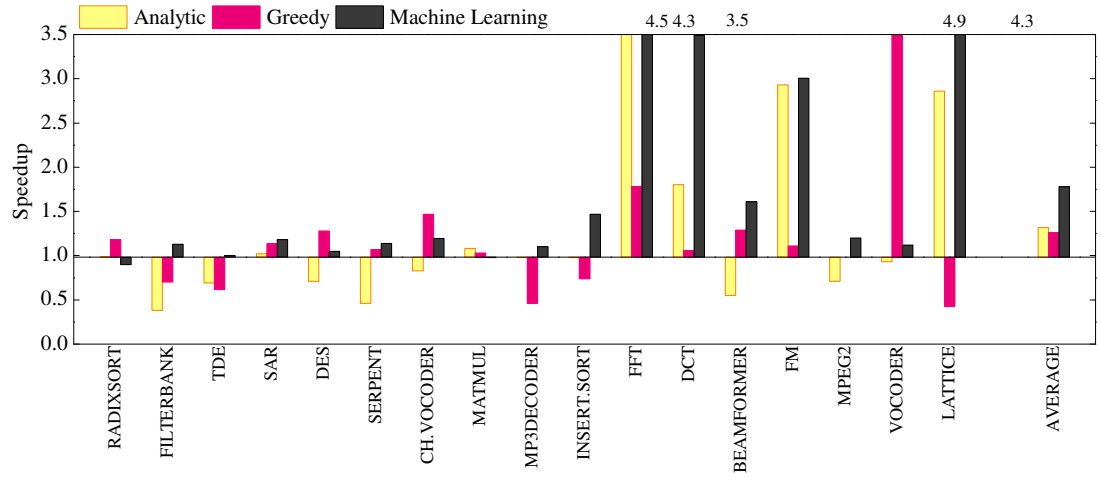


Figure 6.17: Performance comparison on the 8-core platform for the analytical model, the greedy partitioner and the ML-based model.

6.7 Conclusions

This chapter has presented an automatic and portable compiler-based approach to partitioning streaming programs for multi-cores, providing a significant performance improvement over hardwired heuristics. Using machine learning techniques, the compiler predicts the ideal partition structure of a streaming application, allowing a compiler to quickly search the transformation space without running the code. In addition to the predictive model, we have developed a micro-kernel streaming program generator which automatically generates small training examples for the predictive model. We have demonstrated the approach by mapping StreamIt applications onto two multi-core platforms. On average, it achieves a 1.90x speedup over the StreamIt default scheme on a 4-core platform. Compared to a recently proposed analytical-based model, the proposed approach achieves on average a 1.79x performance improvement. When it was ported to an 8-core machine, it is able to achieve a 1.80x improvement over the StreamIt default scheme.

Chapter 7

Conclusions

This chapter summarises the main contributions of this thesis in section 7.1, presents a critical review of this work in section 7.2 and discusses possible directions for future research in section 7.3.

7.1 Contributions

This thesis has studied how to construct efficient and portable techniques to map data and streaming parallelism onto diverse multi-core architectures. As a departure from previous techniques, the work of this thesis has made use of machine learning to automatically construct mapping techniques, which have the potential to simultaneously reduce the human effort required for compiler design and increase the attainable performance of future multi-core systems.

7.1.1 Identifying Profitable Data Parallel Candidates

Chapter 4 has shown that identifying the profitability of data parallel candidates is crucial for achieving an efficient parallel execution. It has also shown that the profitability of a candidate often varies across platforms. Because of this, previous fixed, work-based profitability evaluation heuristics [Hall et al., 1996; William et al., 1996; Tournavitis and Franke, 2009] fail to deliver stable performance when different platforms are targeted.

In contrast to previous approaches, chapter 4 has presented a machine-learning-based, portable approach. Using prior knowledge learnt from the training programs,

the proposed scheme filters out unprofitable parallel candidates while keeping those that are profitable.

Compared with classical, platform-specific approaches [Intel, a; Tournavitis and Franke, 2009], the machine-learning-based scheme is able to achieve much better performance for the NAS parallel benchmark suite on two different multi-core architectures. On a homogeneous multi-core platform (Intel XEON), it achieves almost the same (and sometimes better) level of performance when compared to the manually parallelised code developed by independent experts. On a representative heterogeneous multi-core architecture (IBM CELL), it achieves, on average, a speedup of 9.7 over a state-of-the-art mapping heuristic that is developed by compiler experts [Eichenberger et al., 2005].

Determining the profitability of data parallel candidates is the first step towards efficient data parallelism mappings. Since the proposed model is automatically built from training examples and it uses prior knowledge to make program-aware predictions, the model can adapt to an extensive set of architectures while maintaining robustly high performance across programs.

7.1.2 Determining the Best Number of Threads

Once profitable data parallel candidates have been identified, the next step is to decide how to allocate the hardware resource to the profitable candidates. Chapter 5 has shown that the number of threads allocated to a data parallel candidate has a great impact on its performance.

Chapter 5 has presented two machine-learning-based predictors that predict parallel configurations, i.e. the numbers of threads and the best scheduling policies, for a data parallel candidate across multiple program inputs. On average, both predictors deliver over 96% of the upper-bound performance on the Intel XEON and the IBM CELL platforms. Compared with two state-of-the-art approaches [Blagojevic et al., 2008; Barnes et al., 2008], the machine-learning-based predictors achieve not only better but also more stable performance on the two platforms while having an order of magnitude lower profiling overhead when making predictions for a new program.

With the technique presented in chapter 5, compiler developers are able to automatically build a compiler that can accurately predict the runtime behaviours of data parallel loops across different data sets. As a result, the generated code can automatically adapt to multiple program inputs.

7.1.3 Mapping Streaming Parallelism

Chapter 6 has addressed the problem of partitioning streaming parallelism onto multi-cores. It has shown that the performance of a stream application is largely determined by the way the input program graph is partitioned. It has been observed that the best partitioning strategy varies across different programs and architectures.

Using iterative compiling and executions to find a good partition is not applicable in practice because of the vast number of available partitions available for a single program. Instead, this thesis has presented a two-step, machine-learning-based partitioning approach which does not require any execution of the input program. This approach first predicts the ideal partitioning structure of the input program and then searches the program space (without executing any of the code) to generate a partition that is as close as possible to the prediction. Furthermore, in order to provide sufficient and high-quality training examples, this thesis has also described a synthetic benchmark generator. Using existing stream applications, the benchmark generator can produce many new, diverse training examples which allow supervised learning algorithms to build an accurate predictive model.

Experimental results in chapter 6 show that the machine-learning-based approach significantly outperforms several state-of-the-art partitioning heuristics [Gordon et al., 2006; Navarro et al., 2009; Thies, 2009] and achieves robust performance across programs and architectures.

Unlike prior works on machine-learning-based compilation, the graph partitioning problem concerned in chapter 6 is essentially *unbounded*, i.e. the number of partitioning operations is not fixed. This is the first time a supervised-learning-based model has been built to solve an unbounded compilation problem. Hence, the work presented in chapter 6 offers a new way to solve complicated compilation problems where the optimisation targets are unbounded.

7.2 Critique

This thesis has presented predictive modelling techniques to map data and streaming parallelism onto different multi-core platforms. The optimisation target of this thesis is to reduce execution time of parallel programs and improve state-of-the-art schemes. This goal has been achieved, but there are additional costs that are needed to discuss. This section conducts a critical analysis of this work.

7.2.1 Training Cost

The performance of a supervised-learning-based model heavily depends on the quality of the training data [Bishop, 2006]. No supervised learning tools can create high quality models if there is little to learn from the training examples. Generating training data, of course, can be time-consuming. However, in this thesis all the proposed models were trained off-line and no further training is required once a model has been built. Therefore, training is not a significant cost since it is performed off-line by computers automatically.

Scalability can be another issue for generating training data for new multi-core processors that are likely to have more cores on a single chip. This is particularly true for the stream graph partitioner presented in chapter 6 where on average 3,000 different partitions have been generated for each training program. This number of training examples is sufficient for building an accurate model on the target platforms. But a larger number of partitions for a training program will be necessary when the targeting processor contains more cores. This is because the number of "good" partitions of a program graph is likely to increase as the number of cores increases. Consequently, on these platforms, generating training data might be costly and hereby become a problem. Fortunately, there has been also much research on reducing training costs, via clustering [Thomson et al., 2010] and active learning [Cooper et al., 2006]. The basic idea of these approaches is to generate only new training examples that are sufficiently different from existing ones. This thesis has not used these techniques because the cost of generating training data is not significant on the experimental platforms. Nonetheless, these existing techniques are orthogonal to this thesis.

7.2.2 Profiling Cost

Chapters 4 and 5 have used profiling runs to obtain dynamic features of the target program. Profiling, of course, inevitably introduces additional overhead to the compilation time.

In many application domains, such as embedded systems and consumer applications, reasonably long compile time is often acceptable because the compiled program will run for many times on millions devices and the performance improvement will amortise the compilation overhead. In fact, profiling is a commonly used technique in practice for compiler optimisation. Many approaches, such as auto-tuning frameworks [Dave and Eigenmann, 2009], analytical-based models [Kudlur and Mahlke,

2008] and integer linear programming algorithms [Udupa et al., 2009], all use profiling information to steer the process of optimisation. The techniques presented in chapters 4 and 5 are able to give fairly good performance by using the *smallest* input data set for profiling. According to the experimental results, the proposed techniques have much shorter profiling time than iterative compilation techniques and the profiling-based analytical models. Even so, it would have been desirable to avoid profiling runs at the compilation time. One possible solution is to use just in time (JIT) compilation techniques to gradually collect dynamic information during replication runs of the application [Auslander et al., 1996] so as to achieve a balance between the profiling overhead and the prediction accuracy.

7.2.3 Optimisation Objectives

This thesis has only considered reducing execution time as an optimisation goal. While this is often the main concern in many scenarios, other factors, such as code size and energy consumption can be equally important. Since the proposed models are automatically built from empirical evidence, they can be applied to other optimisation targets as well.

7.2.4 Platforms

One of the advantages of predictive modelling techniques is their ability to be portable across diverse platforms. To demonstrate this merit, all experiments in the thesis have been carried out on two different multi-core platforms. In particular, experiments in chapters 4 and 5 were performed on a homogeneous and a heterogeneous multi-core platforms, while experiments at chapter 6 were performed on two different homogenous multi-core platforms. Because the current implementation of the streaming compiler (i.e. STREAMIT) does not support distributed memory systems, the experiments in chapter 6 were not evaluated on the CELL processor—a distributed memory multi-core. It is preferable to test the model on this platform if there is a implementation for this architecture available, although implementing such a runtime is out of the scope of this thesis.

7.3 Future Work

One important area of future research is that of tuning runtime systems to dynamically map and schedule parallel programs. As many problems in the field of static compilation, the runtime optimisations often imply a vast and complex design space. Finding optimal runtime optimisation options by using iterative search could be too expensive to be performed in practice. This is why many of current runtime systems still use hardwired, target-specific heuristics to make decisions [Stankovic et al., 1995]. They all have the disadvantage of not portable across platforms.

The use of machine learning would, therefore, be beneficial to these systems. It can help the developers to construct runtime optimisation strategies automatically. One could build, for instance, a set of optimisation strategies for the target platform across programs, based on the observations collected on the platform.

Another direction of future research is using predictive techniques to build systems that can dynamically adapt to the changes of program behaviours. Some multi-core designs allow the application to adjust some architecture parameters (e.g. frequency and voltage) at the runtime. By changing the hardware parameters "on-the-fly" according to the dynamic program phases, one can better achieve the optimisation objective, e.g. reduce energy consumption without significantly degrading the application performance.

Machine learning can be used in dynamically configuring the hardware to achieve better runtime adaptation. By combining static program information with observations of the dynamic program execution, one can predict the parallel application's future behaviour. The prediction can be used to guide the configuration of hardware resources. For example, if the next program phase is predicted to be not scale well on the current hardware configuration, a runtime system can set the processor frequency to a lower level (or even turn off some processors) to save energy without slowing down the application too much.

Finally, parallelism mappings can go beyond high level optimisation by considering low-level compiler transformations such as register allocation, loop unrolling and phase reordering. It is well known that high-level optimisation has side effects on the low-level transformation, which may disable some low-level optimisation opportunities [Vegdahl, 1982; Kulkarni et al., 2004]. Therefore, by taking low-level transformations into account when performing high-level parallelism mappings would better than considering each level individually.

Appendix A

Selecting Representative Synthetic Stream Benchmarks

The stream benchmark generator presented in chapter 6 uses the K-Means clustering algorithm to select representative programs from the generated programs. The number of clusters, i.e. the K, is determined by the Bayesian Information Criterion (`BIC`) score which is defined as:

$$BIC_j = \hat{l}_j - \frac{p_j}{2} \cdot logR \tag{A.1}$$

where $\hat{l}_j$ is the log-likelihood of the data when K equals to j, R is the number of points in the data (i.e., the number of generated benchmarks), and p_j is a free parameters to estimate, which is calculated as: $p_j = (K-1) + dK + 1$ for a d-dimension feature vector plus one variance estimate [Sherwood et al., 2002]. $\hat{l}_j$ is computed as:

$$\hat{l}_j = \sum_{n=1}^{K} -\frac{R_n}{2} log(2\pi) - \frac{R_n \cdot d}{2} log(\hat{\sigma}^2) - \frac{R_n - K}{2} + R_n log(R_n/R) \tag{A.2}$$

where R_n is the number of points in the nth cluster and $\hat{\sigma}^2$ is the average variance of the distance from each point to its cluster centre.

We apply the K-Means algorithm to the generated programs with the different cluster numbers K. For each clustering result, its corresponding `BIC` score is computed. The cluster result that gives the best `BIC` score is chosen by the benchmark generator. For each cluster of the chosen result, the benchmark generator selects a number of programs that are close to the cluster centre as representative training programs from this cluster.

Bibliography

A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 308–317, 1988.

Farhana Aleen, Monirul Sharif, and Santosh Pande. Input-driven dynamic execution prediction of streaming applications. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 315–324, 2010.

L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '04, pages 231–239, 2004.

Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 112–125, 1993.

Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, pages 149–159, 1996.

D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks: summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 158–165, 1991.

Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '91, pages 213–223, 1991.

Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski, and Martin Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 368–377, 2008.

Steven Beaty. Genetic algorithms and instruction scheduling. In *Proceedings of the 24th Annual International Symposium*, pages 206–211, 1991.

E. Boser Bernhard, M. Guyon Isabelle, and N. Vapnik Vladimir. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, 1992.

D. Bernstein, M. Golumbic, y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI '89, pages 258–263, 1989.

Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *Int. Conf. on Database Theory*, pages 217–235, 1999.

C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1996.

Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006.

Filip Blagojevic, Xizhou Feng, Kirk Cameron, and Dimitrios S. Nikolopoulos. Modeling multi-grain parallelism on heterogeneous multicore processors: A case study of the Cell BE. In *Proc. of the 2008 International Conference on High-Performance Embedded Architectures and Compilers*, HiPEAC '08, 2008.

Thread Building Blocks. `http://www.threadingbuildingblocks.org/`.

Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '95, pages 207–216, 1995.

Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152, 1992.

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, pages 777–786, 2004.

Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is NP-hard. *Inf. Process. Lett.*, 42(3):153–159, 1992.

J. M. Bull. Measuring synchronisation and scheduling overheads in OpenMP. In *In Proceedings of First European Workshop on OpenMP*, pages 99–105, 1999.

Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-based static branch prediction using machine learning. *ACM Trans. Program. Lang. Syst.*, 19:188–222, 1997.

B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, 2007.

Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

Open Research Compiler. `http://ipf-orc.sourceforge.net`.

Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, LCTES '99, pages 1–9, 1999.

Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steve Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Exploring the structure of the space of compilation sequences using randomized search algorithms. *J. Supercomput.*, 36 (2):135–151, 2006.

Julita Corbalán, Xavier Martorell, and Jesús Labarta. Performance-driven processor allocation. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, OSDI '00, pages 599–611, 2000.

Inmos Corp. *Occam Programming Manual*. Prentice Hall Trade, 1984.

Corinna Cortes and Vladimir Vapnik. Support-vector networks. In *Machine Learning*, pages 273–297, 1995.

Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 157–166, 2006.

Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

Chirag Dave and Rudolf Eigenmann. Automatically tuning parallel and parallelized programs. In *Languages and Compilers for Parallel Computing, 22nd International Workshop*, LCPC '09, pages 126–139, 2009.

E. Culler David, M. Karp Richard, Patterson David, Sahay Abhijit, E. Santos Eunice, Schauser Klaus Erik, Subramonian Ramesh, and Eicken Thorsten von. LogP: a practical model of parallel computation. *Commun. ACM*, 39(11):78–85, 1996. 240477.

Christophe Dubach. *Using Machine-Learning to Efficiently Explore the Architecture/-Compiler Co-Design Space*. Ph.D. thesis, University of Edinburgh, 2009.

Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, Grigori Fursin, and Michael F. P. O'Boyle. Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, Micro 42, pages 78–88, 2009.

Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

Alejandro Duran, Marc Gonzàlez, and Julita Corbalán. Automatic thread distribution for nested parallelism in OpenMP. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 121–130, 2005.

Joseph Earl, Snell Addison, G. Willard Christopher, Tichenor Suzy, Shaffer Dolores, and Conway Steve. Council on competitiveness study of isvs serving the high performance computing market, 2005.

Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for the CELL processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 161–172, 2005.

T. Fahringer. Estimating and optimizing performance for parallel programs. *Computer*, 28(11):47–56, 1995.

Thomas Fahringer, Roman Blasko, and Hans P. Zima. Automatic performance prediction to support parallelization of fortran programs for massively parallel systems. In *Proceedings of the 6th international conference on Supercomputing*, Supercomputing '92, pages 347–356, 1992.

Z. Fang, P. Tang, P.-C. Yew, and C.-Q. Zhu. Dynamic processor self-scheduling for general parallel nested loops. *Computers, IEEE Transactions on*, 39(7):919–929, 1990.

Imola Fodor. A survey of dimension reduction techniques. Technical report, Lawrence Livermore National Laboratory, 2002.

High Performance FORTRAN. `http://hpff.rice.edu/`.

F. Gasperoni, U. Schwiegelshohn, and K. Ebcioğlu. On optimal loop parallelization. In *Proceedings of the 22nd annual workshop on Microprogramming and microarchitecture*, pages 141–147, 1989.

the GNU Compiler Collection GCC. `http://gcc.gnu.org/`.

Marc Gonzàlez, Nikola Vujic, Xavier Martorell, Eduard Ayguadé, Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Tao Zhang, Kevin O'Brien, and Kathryn O'Brien. Hybrid access-specific software cache techniques for the Cell BE architecture. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 292–302, 2008.

Michael I. Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. Ph.d. thesis, Massachusetts Institute of Technology, May 2010.

Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 291–303, 2002.

Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 151–162, 2006.

Ryan E. Grant and Ahmad Afsahi. A comprehensive analysis of OpenMP applications on dual-core Intel Xeon SMPs. In *IEEE International Parallel and Distributed Processing Symposium, 2007.*, pages 1–8, 2007.

Rajiv Gupta, Santosh Pande, Kleanthis Psarris, and Vivek Sarkar. Compilation techniques for parallel systems. *Parallel Comput.*, 25:1741–1783, 1999.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, IEEE International Workshop*, IISWC '01, 2001.

Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih wei Liao, Edouard Bugnion, Monica S. Lain, and Specfp Benchmark. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29:84–89, 1996.

Roger Hoover and Kenneth Zadeck. Generating machine specific optimizing compilers. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 219–229, 1996.

Amir H. Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Flextream: Adaptive compilation of streaming applications for

heterogeneous architectures. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 214–223, 2009.

Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 165–174, 2008.

Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: a method for scheduling parallel loops. *Commun. ACM*, 35(8):90–101, 1992.

IBM. Cell Broadband Engine. `https://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine`.

B Indurkhya, H S Stone, and L Xi-cheng. Optimal partitioning of randomly generated distributed programs. *IEEE Trans. Softw. Eng.*, 12(3):483–495, 1986.

Intel. Intel compilers, a. `http://software.intel.com/en-us/articles/intel-compilers/`.

Intel. Intel new processor generations, b. `http://www.intel.com/pressroom/archive/releases/IntelNewProcessorGenerations.pdf`.

U.J. Kapasi, S. Rixner, W.J. Dally, B. Khailany, Jung Ho Ahn, P. Mattson, and J.D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, 2003.

T. Kisuki, P.M.W. Knijnenburg, and M.F.P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *International Conference on Parallel Architectures and Compilation Techniques.*, PACT '00, 2000.

Toru Kisuki, Peter M. W. Knijnenburg, Michael F. P. O'Boyle, François Bodin, and Harry A. G. Wijshoff. A feasibility study in iterative compilation. In *Proceedings of the Second International Symposium on High Performance Computing*, pages 121–132, 1999.

Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 114–124, 2008.

Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 211–222, 2007.

Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 171–182, 2004.

Kazuhiro Kusano, Shigehisa Satoh, and Mitsuhisa Sato. Performance evaluation of the Omni OpenMP compiler. In *High Performance Computing*, volume 1940, pages 403–414. Springer, 2000.

Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

Mark Van Der Laan and Rine Dudoit. Asymptotic optimality of likelihoodbased cross validation. In *Statistical Applications in Genetics and Molecular Biology*, page 2004, 2003.

C. Lee. UTDSP benchmark suite, 1992. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`.

Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. *Proc. IEEE*, 75 (9), 1987.

Jaejin Lee, Sangmin Seo, Chihun Kim, Junghyun Kim, Posung Chun, Zehra Sura, Jungwon Kim, and SangYong Han. Comic: a coherent shared memory interface for Cell BE. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 303–314, 2008.

Jaejin Lee, Jung-Ho Park, Honggyu Kim, Changhee Jung, Daeseob Lim, and SangYong Han. Adaptive execution techniques of parallel programs for multiprocessors. *Journal of Parallel and Distributed Computing*, 70(5):467 – 480, 2010.

C. Liao and B. Chapman. A compile-time cost model for OpenMP. In *IEEE International Parallel & Distributed Processing Symposium*, IPDPS '07, 2007.

Duo Liu, Zili Shao, Meng Wang, Minyi Guo, and Jingling Xue. Optimal loop parallelization for maximizing iteration-level parallelism. In *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 67–76, 2009.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, 2005.

Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, Micro 42, pages 45–55, 2009.

Raymond Manley and David Gregg. Mapping streaming languages to general purpose processors through vectorization. In *Languages and Compilers for Parallel Computing*, volume 5898 of *LCPC '10*, pages 95–110, 2010.

Henry Massalin. Superoptimizer: a look at the smallest program. In *Proceedings of the second international conference on Architectual support for programming languages and operating systems*, ASPLOS-II, pages 122–126, 1987.

Eliot Moss, Paul Utgoff, John Cavazos, Carla Brodley, David Scheeff, Doina Precup, and Darko Stefanovi. Learning to schedule straight-line code. In *Proceedings of the 1997 conference on Advances in neural information processing systems 10*, pages 929–935, 1998a.

Eliot Moss, Paul Utgoff, John Cavazos, Carla Brodley, David Scheeff, Doina Precup, Darko Stefanovi and #263. Learning to schedule straight-line code. In *Proceedings of the 1997 conference on Advances in neural information processing systems 10*, pages 929–935, 1998b.

Harm Munk, Eduard Ayguadé, and Cédric Bastoul et al. Acotes project: Advanced compiler technologies for embedded streaming. *International Journal of Parallel Programming*, 2010.

Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. Analytical modeling of pipeline parallelism. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 281–290, 2009.

Nils J. Nilsson. Introduction to machine learning: An early draft of a proposed textbook, 1996.

Andy Nisbet. Gaps: A compiler framework for genetic algorithm (ga) optimised parallelisation. In *High-Performance Computing and Networking*, pages 987–989. Springer, 1998.

David L. Olson and Dursun Delen. *Advanced Data Mining Techniques*. Springer, 2008.

David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., 4th edition, 2008.

Karl Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine Series 6*, 2(11):559–572, 1901.

Dan Pelleg and Andrew W. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 727–734, 2000.

Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *Computers, IEEE Transactions on*, C36(12):1425–1439, 1987.

Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: part ii, multidimensional time. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 90–100, 2008.

Zhao Qin, Cutcutache Ioana, and Wong Weng-Fai. Pipa: pipelined profiling and analysis on multi-core systems. In *Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, 2008.

K. Ramamritham and J. A. Stankovic. Dynamic task scheduling in hard real-time distributed systems. *IEEE Softw.*, 1(3):65–75, 1984.

Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 114–123, 2008.

T. Rauber and G. Runger. A transformation approach to derive efficient parallel implementations. *Software Engineering, IEEE Transactions on*, 26:315–339, 2000.

Lawrence Rauchwerger and David Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *ACM SIGPLAN conference on Programming language design and implementation*, PLDI '95, pages 218–232, 1995.

J. L. Rodgers and W. A. Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42:59–66, 1988.

Stuart J. Russell, Peter Norvig, John F. Candy, Jitendra M. Malik, and Douglas D. Edwards. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., 1996.

John Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: optimal vs. heuristic methods in a production compiler. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, pages 1–11, 1996.

P. Sadayappan and Fikret Erçal. Cluster-partitioning approaches to mapping parallel programs onto a hypercube. In *Proceedings of the 1st International Conference on Supercomputing*, ICS '88, pages 475–497, 1988.

Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: concurrent programming for modern architectures. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 271–271, 2007.

V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM J. Res. Dev.*, 35(5-6):779–804, 1991.

Vivek Sarkar and John Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, CC '86, pages 17–26, 1986.

Gideon Schwarz. Estimating the dimension of a model. *Ann. Statist*, 6(2), 1978.

Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 45–57, 2002.

Byoungro So, Mary W. Hall, and Pedro C. Diniz. A compiler approach to fast hardware design space exploration in fpga-based systems. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 165–176, 2002.

J.A. Stankovic, M. Spuri, M. Di Natale, and G.C. Buttazzo. Implications of classical scheduling results for real-time systems. *Computer*, 28(6):16–25, 1995.

Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 123–134, 2005.

Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 77–90, 2003.

Mark W. Stephenson. *Automating the construction of compiler heuristics using machine learning*. Ph.d. thesis, Massachusetts Institute of Technology, 2006.

Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 13–22, 1993.

Jaspal Subhlok, David R. O'Hallaron, Thomas Gross, Peter A. Dinda, and Jon Webb. Communication and memory requirements as the basis for mapping task and data parallel programs. In *Proceedings of the 1994 conference on Supercomputing*, Supercomputing '94, pages 330–339, 1994.

William Thies. *Language and Compiler Support for Stream Programs*. Ph.D. thesis, Massachusetts Institute of Technology, 2009.

William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC' 02, pages 179–196, 2002.

William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Micro 40, pages 356–369, 2007.

John Thomson, Michael O'Boyle, Grigori Fursin, and Björn Franke. Reducing training time in a one-shot machine learning-based compiler. In *Languages and Compilers for Parallel Computing*, LCPC '10, 2010.

Georgios Tournavitis and Bjöern Franke. Atowards automatic profile-driven parallelization of embedded multimedia applications. In *Second Workshop on Programmability Issues for Multi-Core Computers*, MULTIPROG-2009, 2009.

Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 177–187, 2009.

Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Software pipelined execution of stream programs on gpus. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 200–209, 2009.

Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8): 103–111, 1990.

Steven R. Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. In *Proceedings of the 15th annual workshop on Microprogramming*, pages 125–133, 1982.

Michael Voss and Rudolf Eigenmann. Reducing parallel overheads through dynamic serialization. In *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, IPPS '99/SPDP '99, pages 88–92, 1999.

Zheng Wang and Michael F.P. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 75–84, 2009.

Zheng Wang and Michael F.P. O'Boyle. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 307–318, 2010.

Blu William, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with Polaris. *Computer*, 29: 78–82, 1996.

M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *Parallel and Distributed Systems, IEEE Transactions on*, 2(4):452–471, 1991.

Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 274–286, 1996.

Tian Xinmin, Girkar Milind, Shah Sanjiv, Armstrong Douglas, Su Ernesto, and Petersen Paul. Compiler and runtime support for running OpenMP programs on pentium- and itanium-architectures. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, 2003.

Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 63–76, 2003.

Hongtao Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *The 14th International Symposium on High-Performance Computer Architecture*, HPCA '08, pages 290–301, 2008.